June 1985

Report No. STAN-CS-85-1063
Also numbered HPP-84-6

Controlling Recursive Inference

by

David E. Smith
Michael R. Genesereth
Matthew L. Ginsberg

Department of Computer Science

Stanford University
Stanford, CA 94305

Stanford Heuristic Programming Project
Memo HPP-84-6

June 20, 1985

Controlling Recursive Inference

David E. Smith
Michael R. Genesereth
Matthew L. Ginsberg

COMPUTER SCIENCE DEPARTMENT
Stanford University
Stanford, California 94305

Contents

Abstract

Loosely speaking, recursive inference is when an inference procedure generates an infinite sequence of similar subgoals. In general, the control of recursive inference involves demonstrating that recursive portions of a search space will not contribute any new answers to the problem beyond a certain level. We first review a well known syntactic method for controlling repeating inference (inference where the conjuncts processed are instances of their ancestors), provide a proof that it is correct, and discuss the conditions under which the strategy is optimal. We also derive more powerful pruning theorems for cases involving transitivity axioms and cases involving subsumed subgoals. The treatment of repeating inference is followed by consideration of the more difficult problem of recursive inference that does not repeat. Here we show how knowledge of the properties of the relations involved and knowledge about the contents of the system's database can be used to prove that portions of a search space will not contribute any new answers.

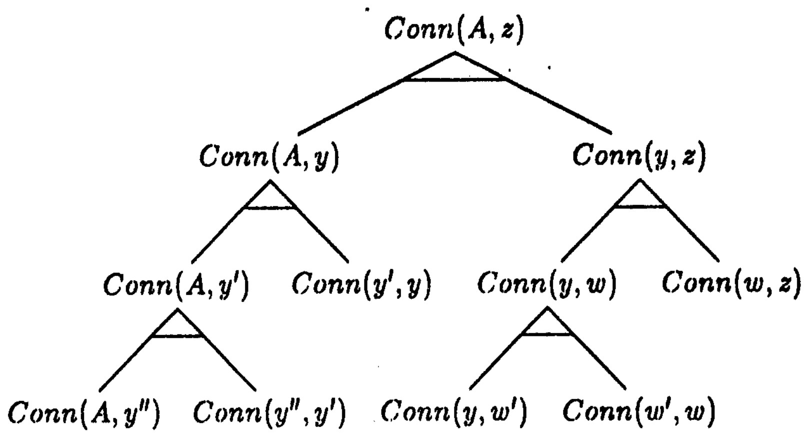

Figure 1-1: A portion of the backward search space for the goal $Conn(A,z)$

1 Introduction

1.1 Motivation

Consider a system for reasoning about circuits based on descriptions of circuit topology and the functional characteristics of circuit elements. Such a system might need to know that connection between terminals in a circuit is transitive and symmetric,

$$\begin{array}{l} Conn(x,y) \wedge Conn(y,z) \Longrightarrow Conn(x,z) \\ Conn(x,y) \Longleftrightarrow Conn(y,x)\ , \end{array} \qquad (1-1)$$

where the proposition $Conn(x,y)$ means the point x is electrically connected to the point y in the circuit. The problem with such facts is that they often result in infinite searches. Suppose, for instance, that we want to find all of the connections to some point A in a circuit. A portion of the backward AND/OR search tree for this problem is shown in Figure 1-1. Applying the transitivity rule to the query $Conn(A,z)$ results in the subgoal $Conn(A,y) \wedge Conn(y,z)$. The transitivity rule can be applied again to both of these conjuncts yielding the subgoals $Conn(A,y') \wedge Conn(y',y)$ and $Conn(y,w) \wedge Conn(w,z)$ respectively. Transitivity applies again to each of these four conjuncts, and so on. For this problem a backward inference procedure would apply the transitivity rule again, and again, and again until it runs out of storage, the user runs out of patience or money, or the machine crashes. The entire space need not be examined in order to find all of the answers to this problem. However, the amount of the search space that must be examined depends upon what connectivity facts are present in the the system's database. In this paper we will consider the problem of how to prune the search space for such recursive problems.

1.2 Cheap Tricks

It might seem that there are simple solutions to the problem given above. Let's consider the possibilities.

1.2.1 Forward Inference

If forward inference were performed on all facts of the form $Conn(a,b)$ using the transitivity rule, and the transitivity rule were not used for backward inference, the problem would be eliminated. Unfortunately, there are several serious difficulties with this approach. First of all, even the restricted use of forward inference can result in the computation and storage of many irrelevant facts. For the example above, forward inference would result in computation of the transitive closure of all connections in the circuit, even though we may only need to know the connections to a few. This would be unacceptable for cases of high fan-out or fan-in, like connections to common busses, power supplies, or grounds. Secondly, as Minker and Nicolas [MN83] and Reiter [Rei78] have pointed out, selective use of forward inference can result in incompleteness in the inference process. For the example above, suppose it were possible to conclude the connectivity of certain points using other axioms. Unless these axioms are also subject to forward inference, the transitive closure of connections derivable by these axioms will not be found. Finally, there are cases where both forward and backward inference can result in infinite search spaces. Consider the rule for computing Fibonacci numbers:

$$\begin{aligned} x = \mathit{Fibonacci}(i-2) \wedge\ y = \mathit{Fibonacci}(i-1) \wedge\ z = x + y \\ \Longrightarrow \mathit{Fibonacci}(i) = z \end{aligned} \qquad (1-2)$$

When two Fibonacci numbers are given to a forward inference procedure it would proceed to compute Fibonacci numbers forever. This rule can cause an infinite loop in either a backward or forward inference engine. Thus, the use of forward inference is not a good solution to the problem of recursive inference.

1.2.2 Eliminating Repeating Goals

This doesn't work in general. For the connectivity example, if the database contains the facts,

$$Conn(A, B)$$
$$Conn(B, C)$$
$$Conn(C, D)$$

it is necessary to search 3 levels deep in the space in order to find all of the connections to A. If repeating goals are pruned, some of the answers will be lost. In Section 3 we will discuss some special cases where this strategy is correct.

1.2.3 Breadth-First Search

Unfortunately, breadth-first search does not help if there are not any solutions to the problem or if the problem involves finding all solutions to a goal. In both cases the entire space must be searched. A complete breadth-first search of an infinite search space is just as bad as a complete depth-first search.

1.2.4 Reformulation

The technique of reformulation is one quite familiar to PROLOG programmers. It involves rewriting the facts available to the inference procedure so that the search space for the goal is no longer infinite, or so that the inference procedure will not discover the recursive portion. As an example of what we mean by *reformulation*, consider the troublesome transitivity rule (1-1) for circuit connections. By introducing a new relation, *IConn*, meaning "immediately connected", the transitivity rule can be rewritten as two separate rules:

$$IConn(x, y) \Longrightarrow Conn(x, y)$$
$$IConn(x, y) \land Conn(y, z) \Longrightarrow Conn(x, z)$$

For this revised set of rules, the search space for the goal proposition $Conn(A, z)$ is shown in Figure 1-2. With this reformulation, we have eliminated the recursion on all left-hand branches of the tree. Of course, for this reformulation to work, all available connections must be expressed in terms of *IConn*, rather than in terms of *Conn*. However, given these relatively minor changes, this revised set of rules will never lead to an infinite search, so long as the *IConn* conjunct is always solved before the *Conn* conjunct.

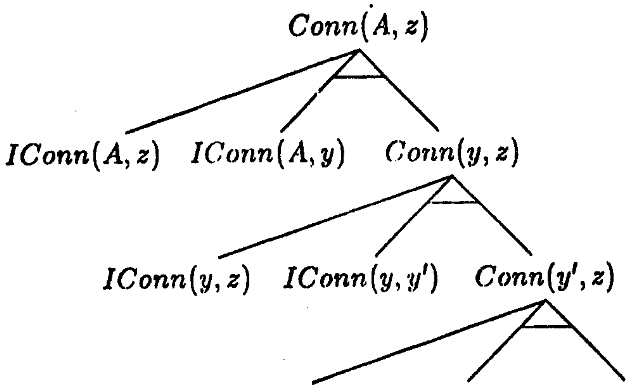

Figure 1-2: Reformulated search space for the goal $Conn(A, z)$

While the above reformulation works well for the query $Conn(A, z)$, it does not work well for the query $Conn(x, D)$. On applying the reformulated version, we would get the subgoal $IConn(x, y) \wedge Conn(y, D)$. If the $IConn$ conjunct is expanded first we end up searching through all of the immediate connections in the circuit; a horribly inefficient process in a large circuit. Alternatively, if the $Conn$ conjunct is expanded first, we again end up with an infinitely repeating search space. The dual reformulation,

$$IConn(x, y) \Longrightarrow Conn(x, y)$$
$$IConn(y, z) \wedge Conn(x, y) \Longrightarrow Conn(x, z)$$

works fine for the query $Conn(x, D)$, but performs miserably for the query $Conn(A, z)$. Neither of these reformulations are reasonable if both kinds of queries are expected, as might be the case for an asymmetric relation. In general reformulations only work effectively for some subset of the possible queries covered by the original domain knowledge.

A second problem with reformulation is that it can be an arbitrarily difficult programming task. Consider the recursive rule that states that a person will be an albino if both his parents are albinos.

$$Albino(x) \wedge Parents(z) = \{x, y\} \wedge Albino(y) \Longrightarrow Albino(z) \qquad (1-3)$$

Suppose that the query is to find all albinos.

$$\textit{find all } z\textit{: } Albino(z)$$

Expanding the parents conjunct first would result in an unacceptable search through all parent/child pairs. Alternatively, expanding either of the albino conjuncts first would result in an infinite repeating search space. By indulging in *knowledge programming*, we could reformulate this rule so that depth-first backward inference results in an efficient search of the space for this query. As a first step, in (1-4) we introduce the new predicate $GivenAlbino(x)$ to refer to those individuals given as albinos initially. The first two rules below state that any given albino is an albino, and that any n^{th} generation descendent of a given albino (along albino lines) is also an albino. We still need to define what it means for an individual to be of albino descent from a given albino. The third rule states that an individual is of albino descent from a given albino if the given albino is a parent of the individual, and the other individual's parent is an albino. The fourth rule simply expresses the transitive closure of this relationship, that an individual is of albino descent from a given albino if that individual is of albino descent from the givens albino children. The final two rules are for checking whether a given individual is an albino and are identical to the original albino rule (1-3).

$$\begin{aligned}
&GivenAlbino(z) \Longrightarrow Albino(z)\\
&GivenAlbino(x) \wedge AlbinoDescent(z,x) \Longrightarrow Albino(z)\\[1ex]
&Parents(z) = \{x,y\} \wedge CheckAlbino(y) \Longrightarrow AlbinoDescent(z,x)\\
&Parents(w) = \{x,y\} \wedge CheckAlbino(y)\\
&\qquad \wedge AlbinoDescent(z,w) \Longrightarrow AlbinoDescent(z,x)\\[1ex]
&GivenAlbino(z) \Longrightarrow CheckAlbino(z)\\
&Parents(z) = \{x,y\} \wedge CheckAlbino(x)\\
&\qquad \wedge CheckAlbino(y) \Longrightarrow CheckAlbino(z)
\end{aligned} \tag{1-4}$$

Performing depth-first backward inference on this reformulation results in forward inference from given albinos to their progeny, and backward inference at each step to verify that the other parent of the progeny is also an albino. Note that this backward portion of the inference is accomplished using the original albino rule (1-3) (which is rewritten using a different predicate to distinguish it from our reformulated version). As with the connectivity example, this reformulation only works efficiently for the query $Albino(z)$, where one or more albinos are desired. It does not work well for checking whether a given individual is an albino.

From these examples, we can see that there are several serious disadvantages to reformulation as a method of controlling recursive inference. First, the resulting knowledge programs only work effectively for some subset of the possible queries covered by the original domain knowledge. Second, it may be an arbitrarily difficult programming task to do such a reformulation. Finally, it is more difficult to understand, explain, and modify reformulations. Reformulation results in an implicit embedding of control information into the domain information. Instead of having facts about the domain and facts about control, the two are merged into knowledge-rich programs for a given interpreter. This has little advantage over building expert systems in more traditional programming languages like LISP or PASCAL. Many authors have argued against reformulation for exactly these reasons [McC68,Hay73,Dav80,Doy80,Cla83,GS85]. The albino reformulation (1-4) leaves much to be desired when compared with the original straightforward domain rule (1-3).

1.3 Definitions

So far we have relied on the readers intuitions and the examples to indicate what we might mean by the term recursive inference. We now give a precise definition.

Let the term *goal set* refer to the set of all conjuncts for a conjunctive goal in a search space. We say that one goal set g' is a ***descendant*** of another goal set g, if there is some sequence of goal sets beginning with g and ending with g', such that each goal set in the sequence is a subgoal of its predecessor. An expression c' is said to be an ***instance*** of an expression c if there is a substitution (a set of bindings) b for the variables in c such that $c' = c|_b$. An ***inference path*** in a search space is a sequence of goal sets in the space such that each goal set in the sequence is an (immediate) subgoal of the preceding goal set. For example, the sequence

$$\begin{array}{l} \{Conn(A,z)\} \\ \{Conn(A,y), Conn(y,z)\} \\ \{Conn(A,y'), Conn(y',y), Conn(y,z)\} \\ \quad\quad\vdots \end{array} \qquad (1-5)$$

is an inference path for the connectivity problem.

Definition 1.1 *An inference path is* recursive *if there is an infinite subsequence* $\langle g_1, \ldots, g_i, \ldots\rangle$ *of the goal sets in the path and a distinguished clause* c_i *in each goal set* g_i *such that,*

1. ***c_i is an instance of c_{i-1}, and***

2. ***c_i is in the subset of g_i that are descendants of c_{i-1}.***

An inference procedure generating any recursive inference path is said to be involved in recursive inference.

The first criterion in the definition restricts us to those paths where the same conjunct is generated over and over again. The second criterion prevents us from considering conjuncts that have not had any inference performed on them yet.

As an example, consider the infinite inference path (1-5) for the connectivity problem. The conjunct $Conn(A, y)$ in the second goal set is an instance of the conjunct $Conn(A, z)$ in the first goal set. The descendants of $Conn(A, z)$ constitute the entire set $\{Conn(A, y), Conn(y, z)\}$, which contains $Conn(A, y)$. Likewise, the conjunct $Conn(A, y')$ in the third goal set is an instance of the conjunct $Conn(A, y)$ in the second goal set. The descendants of $Conn(A, y)$ are the subset $\{Conn(A, y'), Conn(y', y)\}$, which contains $Conn(A, y')$. Thus, with g_i as the i^{th} goal set in the inference path and c_i as the first conjunct in each goal set, the inference path satisfies the definition for a recursive path.

The definition of recursive inference that we have just given actually covers a much broader class of problems than we have considered so far. For example, the definition includes recursive paths where there are intermediate descendants in between those descendants with repeating conjuncts. The definition also includes paths where the repeating conjunct may have its variables bound before it is actually processed. Consider the simple axiom

$$y = x + 1 \ \wedge Integer(x) \Longrightarrow Integer(y) \,. \qquad (1-6)$$

with the query $Integer(2.5)$. One inference path for this problem is shown in Figure 1-3. When the $Integer$ conjuncts are expanded, they are each different, since the variable x is already bound. The subsequence of goals

$$\begin{array}{c} Integer(2.5) \\ Integer(1.5) \\ Integer(.5) \\ \vdots \end{array}$$

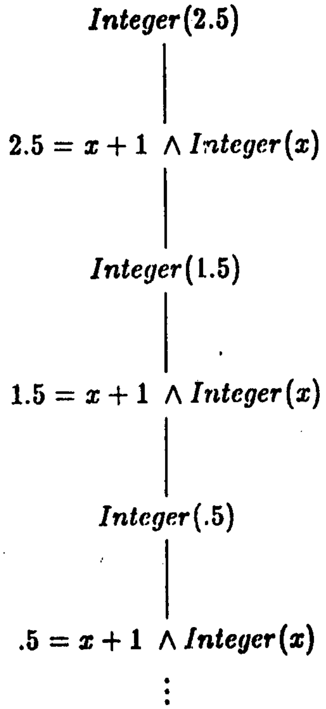
$Integer(2.5)$
$2.5 = x + 1 \wedge Integer(x)$
$Integer(1.5)$
$1.5 = x + 1 \wedge Integer(x)$
$Integer(.5)$
$.5 = x + 1 \wedge Integer(x)$
$\vdots$

Figure 1-3: Inference path for the query *Integer*(2.5).

does not repeat. However, the subsequence

$$\begin{array}{l} 2.5 = x + 1 \ \wedge \mathit{Integer}(x) \\ 1.5 = x + 1 \ \wedge \mathit{Integer}(x) \\ .5 = x + 1 \ \wedge \mathit{Integer}(x) \\ \vdots \end{array}$$

satisfies our definition for recursive inference. Each member contains the conjunct $\mathit{Integer}(x)$, which is both an instance and a descendant of the preceding $\mathit{Integer}(x)$ conjunct.

Although both the integer example and the connectivity example constitute recursive inference, there is an important difference between the two examples. Consider the sequence made up of the conjuncts actually processed by the inference engine at each step. For the connectivity example this sequence repeats.

$$\begin{array}{l} \mathit{Conn}(A,z) \\ \mathit{Conn}(A,y) \\ \mathit{Conn}(A,y') \\ \vdots \end{array}$$

In other words, the repeating conjuncts $\mathit{Conn}(A,\phi)$ are instances of their predecessors at the time they are actually reduced to subgoals. We refer to such recursive inference (where the sequence of conjuncts reduced at each step is repeating) as *repeating inference*.

In contrast, the sequence of conjuncts for the integer example does not repeat.

$$\begin{array}{l} \mathit{Integer}(2.5) \\ 2.5 = x + 1 \\ \mathit{Integer}(1.5) \\ 1.5 = x + 1 \\ \mathit{Integer}(.5) \\ .5 = x + 1 \\ \vdots \end{array}$$

The argument of $\mathit{Integer}(x)$ is always bound before the conjunct is actually reduced to subgoals, and each time the argument is bound to a different constant. The latter qualification is important. It is still possible for the sequence of goals processed to be repeating even though all of their arguments have been bound. We will show examples of this in Section 3. We refer to all non-repeating, recursive inference as *divergent* inference.

1.4 The Approach

Control of recursive inference means eliminating those portions of the search space that are ***superfluous*** or ***redundant***. We say that a goal is ***superfluous*** if there are no facts in the database that will satisfy it or any of its descendants. For a particular problem we say that a goal g is ***redundant*** with another goal g' if none of its descendants will result in any solutions to the problem not produced by descendants to the goal g'. The difficulty is to determine which branches of a search space are indeed superfluous or redundant. If all recursive inference were unproductive it would be a simple matter to provide effective control. However, as we illustrated with the transitivity rule there are many instances where a limited amount of recursive inference is necessary in order to arrive at desired answers. If too much of a recursive space is discarded, important answers to the problem are lost. Alternatively, if not enough of the recursive space is discarded, valuable problem solving effort is wasted.

In general, it is not decidable whether or not a given portion of a recursive search space is redundant. However, there are special cases where it is possible to prove redundancy without completely exploring the space. For repeating inference, a simple syntactic solution is possible. We can decide when to cut off inference by keeping track of the answers produced with each additional level of repetition. For divergent inference the problem is much harder. Here we must generate automatic proofs that no answers exist in a portion of the search space. These proofs are similar to proofs of program termination using well-founded sets. They require information about the properties of the relations involved, and about the content of the system's database. Finally, where rule sets are commutative and each set alone cannot produce answers, it is possible to generate automatic proofs that no novel answers will appear in a portion of the search space, again by making use of knowledge about the properties of the relations involved, and about the contents of the system's database.

1.5 Organization

The next section is a bit of an academic digression. In it, we consider the types of facts that make recursive inference possible, and consider the conditions under which recursive inference will actually occur. The reader more interested in a solution to the problem of recursive inference can skip ahead to Sections 3 and 4. In Section 3, techniques for the common special case of ***repeating inference*** are reviewed. Although several of the algorithms presented are not novel, we consider them from the viewpoint of search control, introduced in Section 1.4. We provide a proof that the method is correct and consider the conditions under which such a strategy is optimal. In addition, more powerful methods for dealing with cases of transitivity and logical subsumption are described.

The more general class of non-repeating recursive inference is considered in Section 4. Here we show how properties of the relations involved and knowledge about the contents of the system's database can be used to demonstrate that a portion of the search space is redundant. Finally, in Section 5 we consider the problem of detecting recursive inference so that control can be instituted only when necessary. Related work is also discussed.

2 The Conditions for Recursive Inference

2.1 Cyclic and Recursive Collections

Suppose that a set of facts can be arranged in the form,

$$\begin{array}{llll} F_1: & L_1 & \wedge \phi_1 & \Longrightarrow L'_2 \\ F_2: & L_2 & \wedge \phi_2 & \Longrightarrow L'_3 \\ & & \vdots & \\ F_{n-1}: & L_{n-1} & \wedge \phi_{n-1} & \Longrightarrow L'_n \\ F_n: & L_n & \wedge \phi_n & \Longrightarrow L'_{n+1} \end{array}$$

where the consequent, L'_{i+1} of each fact F_i unifies with the premise L_{i+1} in its successor F_{i+1}, and the consequent, L'_{n+1}, of the final rule F_n unifies with the premise L_1 in the first rule F_1. We say that such a set of rules forms a ***cycle*** and constitutes a ***cyclic collection.***[1] For example the set of rules,

$$\begin{array}{c} P(x) \Longrightarrow Q(x) \\ Q(B) \Longrightarrow P(A) \end{array}$$

form a cycle, since $Q(x)$ unifies with $Q(B)$ and $P(x)$ unifies with $P(A)$. A rule can be involved in more than one cycle, so we also refer to the union of any two cyclic collections that share rules as a cyclic collection.

If a common set of bindings is possible for all of the unifications in a cycle, the facts are said to be ***recursive*** and constitute a ***recursive collection.*** In other words, a group of facts is recursive if there is some common set of bindings b for the variables in each of the facts F_1 through F_n such that $L_i|_b = L'_i|_b$ and $L'_{n+1}|_b = L_1|_b$. (The notation $P|_b$ refers to the clause P under the variable bindings b). The cyclic collection given above is not a recursive collection because x cannot be bound to both A and B simultaneously. However, the cyclic collection

$$\begin{array}{c} P(x) \Longrightarrow Q(x) \\ Q(y) \Longrightarrow P(y) \end{array}$$

[1] This notation and terminology is derived from Minker and Nicolas [MN83]. Minker and Nicolas express these definitions in terms of facts in conjunctive normal form. For simplicity we have expressed these definitions in terms of rules.

is a recursive collection since the binding $x : y$ unifies $Q(x)$ with $Q(y)$ and $P(x)$ with $P(y)$.[2] Likewise, the transitivity and symmetry rules, the albino rule, and the Fibonacci rule given in the previous section are all recursive collections.

As with cyclic collections, it is possible for a single rule to be part of more than one recursive collection. We therefore refer to the union of any two recursive collections that share rules as a recursive collection.

2.2 Recursive Search Spaces

We say that a search space is ***recursive*** if it contains a recursive inference path (as defined in Section 1.3). It should come as no surprise that recursive collections give rise to recursive search spaces.

Theorem 2.1 *For any recursive collection of facts there is at least one goal that will result in a recursive search space.*

Proof *By the definition of a recursive collection, there is some common set of bindings b for the variables in each of the facts F_1 through F_n such that each $L_i|_b = L'_i|_b$ and $L'_{n+1}|_b = L_1|_b$. From the goal proposition $L_1|_b$, using the rules $R_1, ..., R_n$ it is possible to regenerate the subgoal $L_1|_b$. This process can be repeated arbitrarily many times, resulting in an arbitrarily long inference path.* □

As an example, consider the transitivity rule for circuit connections. The search space in Figure 1-1 shows that the goal $Conn(A, z)$ has a recursive space since each subgoal in the sequence

$$\langle Conn(A, z), Conn(A, y), Conn(A, y'), Conn(A, y''), \ldots \rangle$$

is an instance of the preceding goal.

Corollary 2.2 *If a goal proposition g results in a recursive search space for a given recursive collection, any generalization g' of the goal will also result in a recursive search space.*

By a generalization of a proposition g, we mean a proposition g' such that $g'|_b = g$ for some set of bindings b. For example, since the query $Conn(A, z)$

[2] This is not a very interesting recursive collection. Most recursive collections have additional conjuncts in their premises.

has a recursive search space, the query $Conn(x, z)$ will also have a recursive search space.

It is natural to ask whether recursive collections are the only kinds of facts that can lead to infinite search spaces. Infinite search spaces can always occur if there is an infinite database, but barring this possibility, the answer appears to be yes.

Conjecture 2.3 *If an infinite path exists in the search space there must be a recursive collection of facts involved in the generation of that path.*

In fact, we believe that a stronger statement holds.

Conjecture 2.4 *A set of n axioms that is not a recursive collection can generate an inference path of at most length* $(2^n - 2)a + 1$ *where a is the maximum arity (number of arguments) of all relations in the collection.*[3]

Lewis [Lew75] has proven a weaker theorem, but we are not aware of any proof of these conjectures.

2.3 Recursive Inference

As we stated in Section 1.3, recursive inference occurs when an inference procedure follows a recursive path in a recursive search space. By this definition a recursive search space is a necessary condition for recursive inference, but it is not a sufficient condition. Even though a given problem may have a recursive search space, recursive inference will not necessarily result. Consider the reformulated version of the transitivity axiom for circuit connections (Section 1.2.4). Although the search space for the goal $Conn(A, z)$ is still a recursive space, if the $IConn$ conjunct is always solved first, recursive inference will never result for this goal.

In general, whether or not recursive inference occurs depends upon

- the specific characteristics of the recursive collections involved,
- the search strategy employed by the inference procedure,

[3] We arrived at the formula $(2^n - 2)a + 1$ by empirical generalization of a set of examples, beginning with the cyclic collection

$$P(y, x) \Longrightarrow Q(x, y)$$
$$Q(A, x) \Longrightarrow P(B, x)$$

and progressing to higher arity, more rules and rules involving functional expressions.

- the strategy for evaluating embedded functional expressions (whether they are evaluated or treated syntactically),
- the number of answers desired for the problem, and
- the number of answers actually available for the problem.

If non-recursive subgoals are preferred to recursive subgoals, the chances of recursive inference will be less. Likewise, if non-recursive clauses are preferred to recursive clauses in conjunctive subgoals, the chances of recursive inference will be less. In contrast, recursive inference becomes more likely as the ratio of number of solutions sought to number of solutions available increases. Unfortunately, there is no simple precise characterization of when recursive inference will or will not occur. Any such characterization would require a classification of all the different possibilities for each factor, and a multi-dimensional table to consider all of the different combinations.

Since some of the factors are uncontrollable (or the cure is worse than the disease), the best that we can say is that when a recursive collection is present, there is the potential for recursive inference.

3 Repeating Inference

As indicated in Section 1.3, repeating inference is when the sequence of processed goal conjuncts actually repeats. In other words, there is some infinite subsequence of the goal conjuncts processed, such that each successive conjunct is an instance of its predecessor. Most of the examples that we considered in Section 1 were of repeating inference. In particular, the connectivity example had this characteristic, since a goal expression of the form $Conn(A, z)$ is generated and expanded repeatedly in the leftmost branch of the AND/OR search tree.

In repeating inference, a portion of the AND/OR search space is repeated over and over again. To control the search we must determine the level at which the repetition can be cut off. The search space below the cutoff point must not hold any new answers.

3.1 Finding a Single Answer

First consider the special case where only a single answer is needed for a query. In such cases, if an answer cannot be found without exploring a repeating portion of the space, no answer can be found at all. As a result, the search space can be pruned drastically.

Theorem 3.1 *If only a single answer is needed for a goal g, any descendant g' that is an instance of g can be discarded (along with the entire subspace descending from g'). Furthermore, it is optimal to do so (i.e. the least expensive way of finding an answer does not involve searching the subspaces descendant from any of the g').*

Some notation is needed in order to demonstrate this result and other results to follow. Let $S(g)$ refer to the search space beginning with the goal g and containing all of the legal descendants of the goal g. A ***frontier set*** F of a search space $S(g)$ is defined to be a set of goals in the space such that no goal in the frontier set is a descendant of any other goal in the frontier set. Intuitively, a frontier set is some possibly jagged, partial slice through a search space. Let $S_F(g)$ refer to that portion of the space $S(g)$ not including any of the frontier goals $f \in F$ or their descendants $S(f)$. In other words, the restricted search space $S_F(g)$ is just $S(g)$ with all of the frontier branches pruned out. Let $A_F(g)$ refer to the set of answers to the goal g present in the restricted space $S_F(g)$.

For a recursive space let $R_n(g)$ refer to the frontier set consisting of the n^{th} level repetitions of the goal g. Using this notation the above theorem can be stated more precisely as,

> *If only a single answer is needed for the goal g, the optimal strategy will involve a search in $S_{R_1(g)}(g)$.*

Proof *Suppose that the space $S_{R_1(g)}(g)$ contains no answers to g. Since each goal $g' \in R_1(g)$ is an instance of g, the space $S_{R_1(g')}(g')$ does not contain any answers to g'. By induction, there are no answers to any of the repeating goals, and hence there are no answers to g. Consequently, restricting the search to $S_{R_1(g)}(g)$ will still result in the correct answer and will not sacrifice optimality.*

Suppose, on the other hand, that there is an answer to g in $S_{R_1(g)}(g)$. Let α' be the easiest answer to find for the goal $g' \in R_1(g)$. The same proof will result in an answer α to the goal g. Therefore the answer α will be easier to find for g than the answer resulting from α'. As a result, restricting the search to $S_{R_1(g)}(g)$ will not sacrifice optimality. □

There is a useful corollary of this theorem.

Corollary 3.2 *Repeated ground queries and functional queries can always be pruned from a search space.*

This result holds because functional queries never have more than one answer.

Finally, note that Theorem 3.1 does not mean that all repetitions can be discarded, only those for goals that require only one solution. Consider the hypothetical search space in Figure 3-1. The goal g, which has only a single solution, generates a conjunctive descendant $h \wedge j$. It might be necessary to search through several of the answers to the conjunct h in order to find a solution to the conjunction. Thus, while any reoccurences of g can be discarded, reoccurences of the goal h cannot be.

3.2 Finding Multiple Answers

In cases where more than one answer is needed, Theorem 3.1 does not apply. Such problems arise far more often than might be expected. Even though only a single answer is needed for a problem, some of its subproblems may be conjunctive, as in the example above. Solving a conjunction frequently requires generating more than one solution to at least one of the conjuncts.

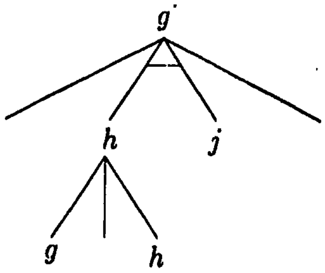

Figure 3-1: Search space for a single-solution problem

3.2.1 The Theory

If multiple answers are needed, in order to eliminate a portion of the repeating space we must show that that portion of the space is ***redundant*** (i.e. will not produce any novel answers to the original problem). What makes such a proof possible is the observation that if a search of one or more levels of repetition deeper in a recursive space does not produce any new answers, no amount of additional search will produce any new answers to the original repeated supergoal. Using the notation introduced in the previous section, we can state this more precisely.

Theorem 3.3 *Let F be the frontier set $R_n(g)$ consisting of n^{th} level instances of the goal g. Let F' be a frontier set consisting of repeating descendants of goals in the set F. If $A_{F'}(g) = A_F(g)$, all of the frontier subspaces $S(r)$ for $r \in F$ are redundant.*[4]

Proof *The proof is by induction on the level of repetition in the search space. First we prove the theorem for the case where $F' = R_{n+1}(g)$. Let g' be a first level repeating descendant of g and let b be the set of bindings such that $g' = g|_b$. Let r be the subset of $R_n(g)$ that are descendants of g' as shown in Figure 3-2. Thus $r = R_{n-1}(g')$. Let r' be the set $R_1(r) = R_n(g')$ (all first*

[4] This theorem relies on the assumption of complete indexing in the problem solver's database. In other words, the system must be able to find any fact in the database that matches a goal. Without complete indexing, answers could be found to an instance of a goal when they could not be found for the original goal. A weaker version of the theorem still holds if complete indexing of the problem solver's database is not assumed. In this case, the frontier set F must contain repetitions instead of instances of the goal g. Essentially, this means that search must be a few recursion levels deeper until a specialization of the initial goal is found for which F will contain pure repetitions.

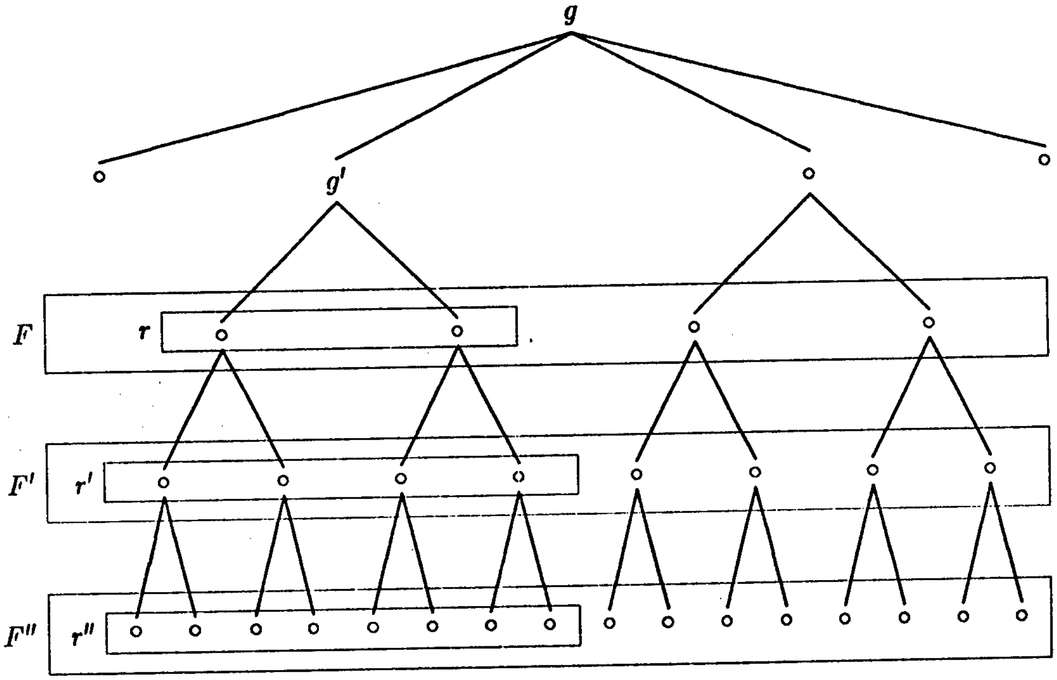

Figure 3-2: Abstract repeating search space.

level repeating descendants of r) and let r'' be the set $R_2(r) = R_{n+1}(g')$ (all second level repeating descendants of r).

The space $S_{r'}(g')$ is an instance of a portion of the space $S_F(g)$. In fact,

$$A_{r'}(g') = \{a : a \cup b \in A_F(g)\} \ .$$

Likewise the space $S_{r''}(g')$ is an instance of a portion of the space $S_{F'}(g)$ so

$$A_{r''}(g') = \{a : a \cup b \in A_{F'}(g)\} \ .$$

Since $A_F(g) = A_{F'}(g)$, it follows that $A_{r'}(g') = A_{r''}(g')$.

Let $\phi_{g'}$ be the function from answers to a descendant $g' \in R_1(g)$ to answers to the supergoal g. In other words, $a = \phi_{g'}(a')$ means that if a' is an answer to g', a will be an answer to g. Then,

$$A(g) = A_{g'}(g) \cup \{a : a = \phi_{g'}(a') \wedge a' \in A(g')\} \ .$$

Now consider the frontier $F'' = R_{n+2}(g)$. Using the two results above,

$$\begin{aligned} A_{F''}(g) &= A_{R_1(g)}(g) \cup \bigcup_{g' \in R_1(g)} \{a : a = \phi_{g'}(a') \wedge a' \in A_{r''}(g')\} \\ &= A_{R_1(g)}(g) \cup \bigcup_{g' \in R_1(g)} \{a : a = \phi_{g'}(a') \wedge a' \in A_{r'}(g')\} \\ &= A_{F'}(g) \ . \end{aligned}$$

By induction $A_{F^{(k)}}(g) = A_F(g)$ for all k. Thus, $A(g) = A_F(g)$, which means that the repeating descendants in F are redundant.

Finally, for any set F' satisfying the requirements of the theorem, $S_{R_{n+1}(g)}(g) \subseteq S_{F'}(g)$, so $A_{F'}(g) = A_F(g)$ implies that $A_{R_{n+1}(g)}(g) = A_F(g)$. Since the theorem holds for $F' = R_{n+1}(g)$ it holds for arbitrary F'. □

Corollary 3.4 *The depth of repetition in a search space can be limited to one less than the total number of answers desired for the problem.*

Theorem 3.1 is a special case of this corollary.

Example: Consider the connectivity axiom for circuits,

$$Conn(x, y) \wedge Conn(y, z) \Longrightarrow Conn(x, z) \ .$$

As before, suppose that the problem is to find all points in a circuit connected to a given point A,

find all z: $Conn(A, z)$.

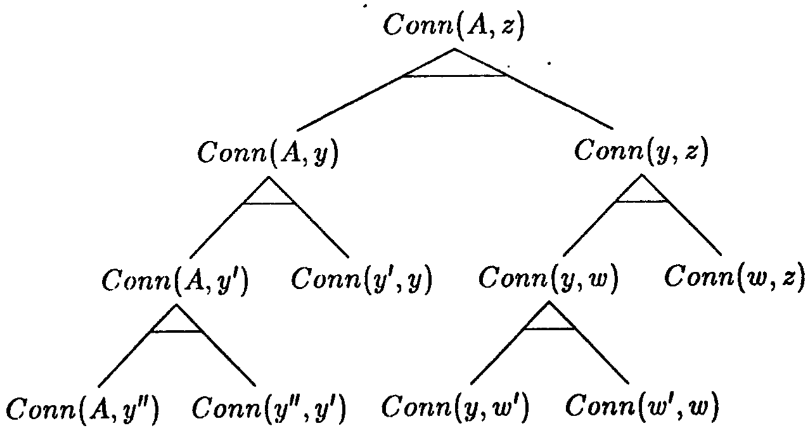

Figure 3-3: A portion of the backward search space for the goal $Conn(A, z)$

An initial portion of the backward AND/OR search space for this problem is reproduced in Figure 3-3. If there are no answers in the system's database for $Conn(A, z)$, there are no answers at all. In this case the frontier sets $F = \{\text{``}Conn(A, z)\text{''}\}$ and $F' = \{\text{``}Conn(A, y)\text{''}\}$ satisfy the conditions of Theorem 3.3. $S_F(g)$ is the null space and $S_{F'}(g)$ is the space consisting of only the goal $g = \text{``}Conn(A, z)\text{''}$. Since there are no answers in the database for the goal g, $A_F(g) = A_{F'}(g) = \emptyset$. As a result, Theorem 3.3 states that no search is necessary for the problem.

If, instead, the database contains the fact $Conn(A, B)$, but no facts about the connections to B, the sets

$$F = \{\text{``}Conn(A, y)\text{''}\}$$

and

$$F' = \{\text{``}Conn(A, y')\text{''}\}$$

satisfy the theorem. In this case, $S_F(g)$ and $S_{F'}(g)$ both contain the single answer $z = B$. As a result, only database answers to the initial goal $Conn(A, z)$ need be located in this case.

Finally, suppose that the database contains the facts $Conn(A, B)$ and $Conn(B, C)$ but no other connections to A, B or C. For this case, the cutoff frontiers contain two terms since the right hand branch of the conjunction also contributes a recursive branch for the binding $z = B$.

$$F = \{\text{``}Conn(A, y')\text{''}, \text{``}Conn(B, w)\text{''}\}$$

$$F' = \{\text{“}Conn(A, y'')\text{”}, \text{“}Conn(B, w')\text{”}\}$$

For both of these frontiers, the answer set will consist of $z = B$ and $z = C$.

Optimality Although Theorem 3.3 tells us some conditions under which a portion of the search space is redundant, it does not tell us that pruning the redundant portion of the space is necessarily optimal. In some cases it can be advantageous to search part of the redundant portion of the space. As an example, consider the connectivity example of the previous section, where the available facts were $Conn(A, B)$ and $Conn(B, C)$. Suppose that we also have the (non-recursive) collection of facts

$$\begin{aligned} H(x, y) &\Longrightarrow Conn(x, y) \\ G(x, y) &\Longrightarrow H(x, y) \\ F(x, y) &\Longrightarrow G(x, y) \\ E(x, y) &\Longrightarrow F(x, y) \end{aligned}$$

together with the facts $E(A, B)$ and $E(A, C)$. In this case, we could find all the answers to the query $Conn(A, z)$ by exploring this non-recursive path. Theorem 3.3, therefore, allows us to conclude that the goal $Conn(A, y)$ is redundant. However, the non-recursive way of finding the answer $z = C$ is longer and more costly than finding the same answer by exploring a level deeper in the repeating space. As a result, pruning the subgoal $Conn(A, z)$ is non-optimal for this case.

In the case where all of the solutions are needed to a problem, we can show that pruning the redundant portion will be optimal.

Theorem 3.5 *For recursive problems where all of the solutions are sought, if there exists a frontier F that obeys the conditions of Theorem 3.3 the optimal strategy will involve searching only $S_F(g)$.*

Proof *In order to find all answers in a space, all portions that may contain novel answers must be searched. Assuming that we do not know which portions of $S_F(r)$ are redundant with $S(r)$ for each $r \in F$ then, $S_F(g)$ must be searched in any case. If $S_F(g)$ must be searched, each of the $S(r)$ contain only redundant answers, so there is no advantage to searching any of them. As a result, the optimal strategy will be a search over only $S_F(g)$.* □

As we demonstrated in the example above, this result does not hold for problems where some specific number of solutions is sought.

3.2.2 Repetition Cutoff Algorithms[5]

In order to make use of Theorem 3.3 we need a mechanism for finding the repetition level that satisfies the conditions of the theorem. Finding such a frontier set requires preserving the answers to any goal with repeating descendants.

Algorithm 3.6

1. If a goal g_i is an instance of one of its supergoals g, the goal g_i is suspended until all other alternatives for solving g have been exhausted.
2. If any new answers are found to the goal g, all repeated instances g_i of g are enabled for another level of expansion. If not, the inference is terminated.

A flowchart of a problem solver incorporating this procedure appears in Figure 3-4. One major efficiency improvement can be made on this procedure. The answers produced by expanding the search space an additional level of repetition will be a subset of those produced in the first level since each repeated descendant g_i is an instance of the goal g. Therefore, it is not necessary to reproduce the space at each level. It is sufficient to cache all of the answers to the supergoal and use them as the answers to any repeated descendants. Thus, a more efficient procedure would be

Algorithm 3.7

1. Each time a solution is found to a query (or subquery) the solution is cached.
2. When a repeated descendant is encountered, only instances found in the system's database (including cached answers) are used as solutions to the repeated descendant. No additional inference is performed on this repeated descendant.
3. The solution of a repeated descendant is not complete until no additional solutions can be found to the goal that it is a repeat of. In other words, new answers to a goal must continually be plugged into all repeated descendants until quiescence occurs and no new answers appear.

[5] These algorithms were first discovered by Black [Bla68] and were later rediscovered by McKay and Shapiro [MS81] and by the authors.

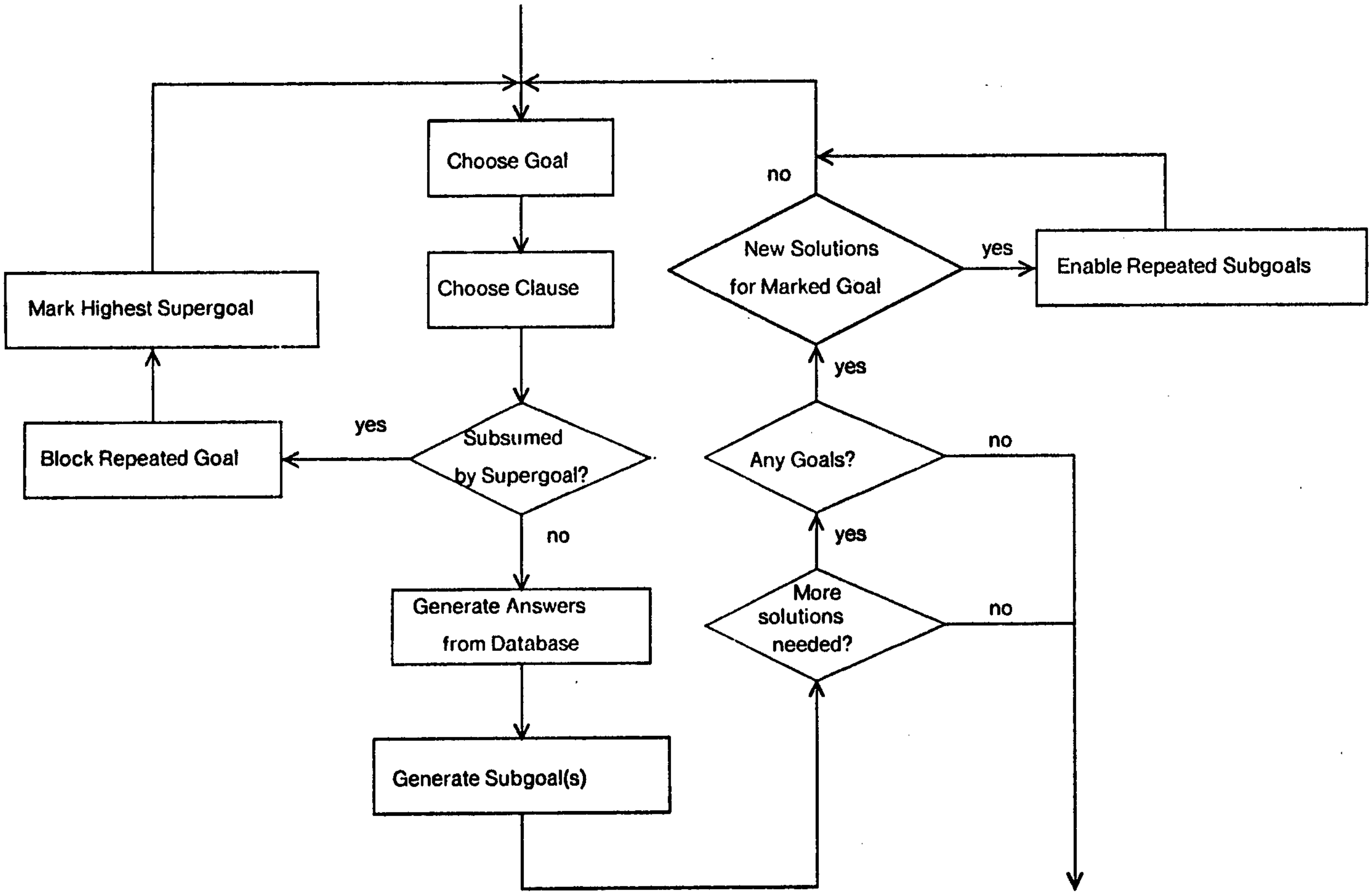

Figure 3-4: Backward inference procedure with repetition control.

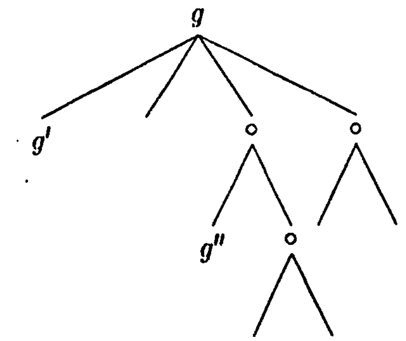

Figure 3-5: An idealized AND/OR tree containing repetition

As an illustration of this method, consider the search tree shown in Figure 3-5. This tree is a snapshot of the goal stack for the inference engine at some point in the computation. There are two repetitions of the original goal expression g, both of which are suspended awaiting answers to g. If the answer α is found to g, this answer is cached and consequently plugged in as an answer for g' and g''. If these branches generate additional answers α' and α'' to g, then these answers, in turn, are cached and must be tried in the two repeated descendants. When no new answers to g can be produced the process is complete.

Example: Circuit Connections Consider the connectivity example of the Section 3.2.1 where the goal is to find all points in a circuit connected to a given point A and the database contains the facts $Conn(A, B)$ and $Conn(B, C)$. First the answer $z : B$ is found. The transitivity rule is then applied to the initial goal yielding $Conn(A, y) \wedge Conn(y, z)$. Since the clause $Conn(A, y)$ is an instance of the initial goal, $Conn(A, z)$, no inference is performed on this clause. However, since there is already a cached solution to the original goal $Conn(A, z)$ the solution $y : B$ is found for the repeated descendant. Substituting this binding into the other conjunct yields the subgoal $Conn(B, z)$. The answer $z : C$ is found in the system's database and is therefore cached as a solution to the original goal. The descendant $Conn(B, z)$ is then expanded using the transitivity rule, yielding the conjunction $Conn(B, y') \wedge Conn(y', z)$. As with the first expansion, the clause $Conn(B, y')$ is an instance of the subgoal $Conn(B, z)$ so no inference is performed on the repeated clause $Conn(B, y')$. As before there is

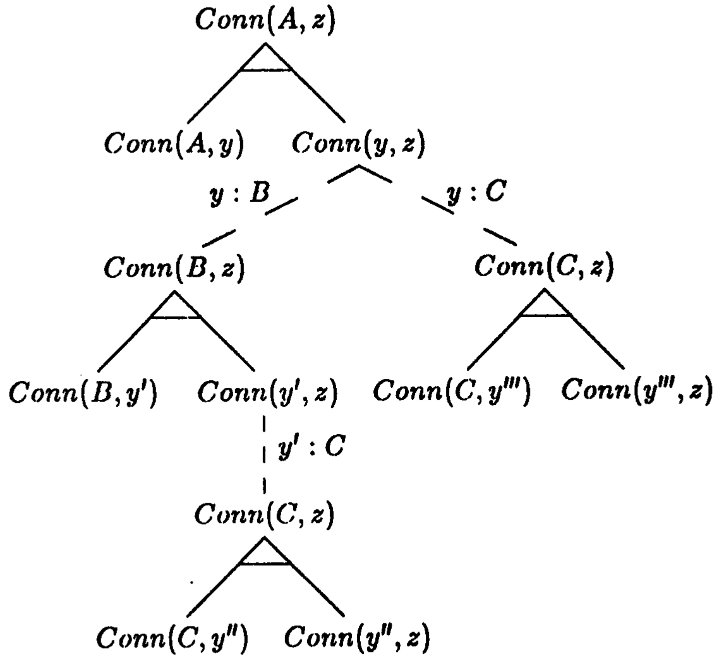

Figure 3-6: Search for the query $Conn(A, z)$

already a solution, $y' : C$, in the system's database to the repeated clause. This solution is substituted into the other conjunct yielding the subgoal $Conn(C, z)$. There are no solutions to this clause in the system's database. The expansion to $Conn(C, y'') \wedge Conn(y'', z)$ again contains the repeating subgoal $Conn(C, y'')$, so no further inference is performed and no answers are found to $Conn(C, z)$. This leaves no further alternatives for the supergoal $Conn(B, z)$. However, this subgoal did generate an additional answer $(z : C)$ to the initial goal $Conn(A, z)$, so the cached fact must be used in the first repeating descendant $Conn(A, y')$. Substituting the binding $y' : C$ into the other conjunct yields the subgoal $Conn(C, z)$. As before, this subgoal yields no solutions, and the inference process terminates.

Example: Ancestry As a second example, consider the problem of finding all albinos, given the rule

$$Albino(x) \wedge Parents(z) = \{x, y\} \wedge Albino(y) \Longrightarrow Albino(z)$$

Assume that our database contains the facts:

$$\begin{aligned}&Parents(ABCD) = \{AB, CD\}\\&Parents(AB) = \{A, B\}\\&Parents(CD) = \{C, D\}\end{aligned}$$

$$\begin{aligned}&Albino(A)\\&Albino(B)\\&Albino(C)\\&Albino(D)\end{aligned}$$

Beginning with the conjunct $Albino(z)$ the system would first discover the four answers in its database. It would then apply the recursive rule resulting in the conjunction $Albino(x) \wedge Parents(z) = \{x, y\} \wedge Albino(y)$. The first of these conjuncts is identical to its parent so the algorithm would halt further inference on this branch. However, since there are already four cached solutions to the original problem, these are substituted in as solutions to the repeated descendant. We are left with the conjunction $Parents(z) = \{x, y\} \wedge Albino(y)$ for the cases of $x = A$, $x = B$, $x = C$ and $x = D$. For

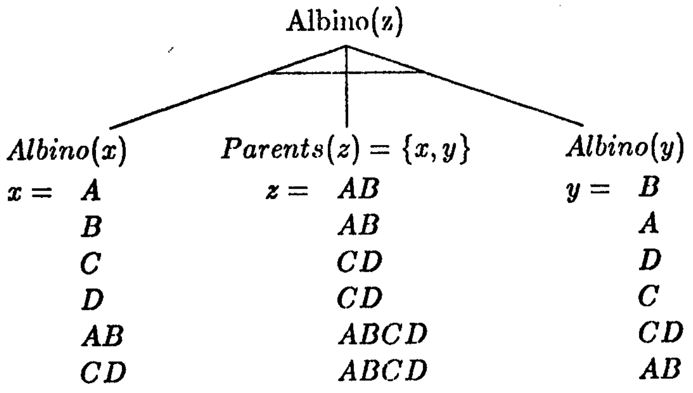

Figure 3-7: Search space for the query *Albino*(z).

these different bindings, the parents conjunct yields values for y and z:

x	y	z
A	B	AB
B	A	AB
C	D	CD
D	C	CD

For each of these solutions for y, the final conjunct $Albino(y)$ is verified by reference to the database. Thus, the answers AB and CD are produced and cached for the original query. These, in turn, are substituted into the repeated descendant, again yielding the conjunction $Parents(z) = \{x, y\} \wedge Albino(y)$ for the cases $x = AB$ and $x = CD$. The parents conjuncts yield new values for y and z:

x	y	z
AB	CD	$ABCD$
CD	AB	$ABCD$

Again the conjuncts $Albino(CD)$ and $Albino(AB)$ are verified by reference to the database so the answer $ABCD$ is produced and cached for the original query.

Finally, substitution of $ABCD$ into the repeated descendant yields no additional answers (since $ABCD$ has no progeny) and the search terminates. A sketch of this search space appears in Figure 3-7.

Soundness, Completeness and Optimality

Since Algorithms 3.6 and 3.7 are merely ways of finding frontier sets that satisfy Theorem 3.3, they do not adversely affect the logical soundness or completeness of an inference procedure. Although use of these algorithms will result in a drastic reduction of the size of repeating search spaces, their use does not guarantee termination of search. This is because the restricted search space $S_R(g)$ can still be infinite. Theorem 3.3 and the algorithms do not detect or eliminate divergent inference paths. As a result, an inference procedure making use of the cutoff algorithms might still encounter a divergent path and might therefore never terminate or find all of the answers in the space.

However, barring divergent paths, inference procedures based on Algorithms 3.6 and 3.7 are guaranteed to terminate, and any solution in the search space will be found. If the subgoal generator is logically complete, such an inference procedure will also be logically complete.

The algorithms limit search to one recursion level beyond the minimal level that satisfies the conditions of Theorem 3.3. In Section 3.2.1 we pointed out that cutting off redundant recursive paths at the earliest possible level may not be optimal. This observation therefore extends to the algorithms as well.

3.3 Special Types of Repetition

The theorems of the previous section are general, but weak. There are special cases of repeating inference where stronger results are possible. For example, in Section 3.1 we developed a much stronger result for the case where only a single answer was needed. There are two other special cases that merit particular attention, descendant subsumption, and transitivity. Both of these cases rely on general *goal subsumption* for their power. We say that a goal set g *subsumes* another goal set g' if there is a set of bindings b such that $g|_b \subseteq g'$. If only a single answer is needed for a problem any goal set g' subsumed by another goal set g will be redundant with that descendant [Nil80]. The situation is somewhat more complicated when more than one answer is needed.

Theorem 3.8 *Let g' and g'' be descendants of g and let b' and b'' be the binding sets that relate solutions to g' and g'' to solutions to g (i.e. $g' \Longrightarrow g|_{b'}$ and $g'' \Longrightarrow g|_{b''}$). Suppose that g' subsumes g'' with the bindings c. If the bindings for the output variables in $b' \cup c$ are a subset of the bindings b'', g'' is redundant with g'.*

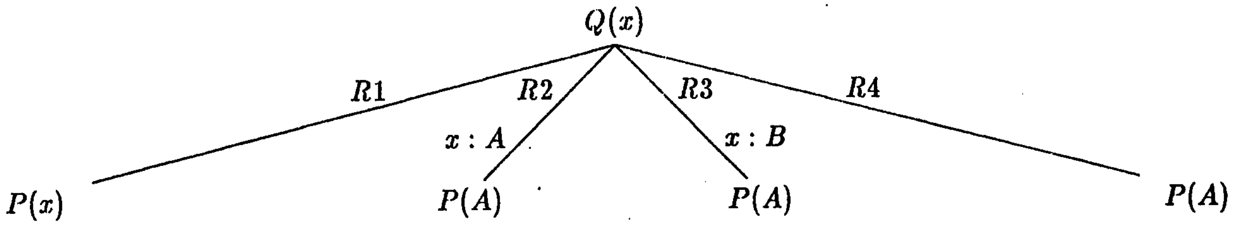

Figure 3-8: Subsumption example

Proof *Let α'' be a solution found to g''. Then $\alpha'' \cup b''$ is a solution to g that will be found by exploring g''. We must show that the same solution (or a generalization of it) can be found by exploring g'. Since $g'|_c \subseteq g''$ we know that there is some solution $\alpha' \subseteq \alpha'' \cup c$ that will be found for g' (assuming complete database indexing). Thus, $\alpha' \cup b'$ will also be a solution to g. But since $\alpha' \subseteq \alpha'' \cup c$ and $b' \cup c \subseteq b''$ we have $\alpha' \cup b' \subseteq \alpha'' \cup c \cup b' \subseteq \alpha'' \cup b''$. So the solution $\alpha' \cup b'$ found by the descendant g' is a generalization of the solution $\alpha'' \cup b''$ found by the descendant g''. g'' is therefore redundant with g'.* □

As an example, consider the simple search space of Figure 3-8 generated from the goal $Q(x)$ using the rules

$$\begin{array}{rl} R1: & P(x) \Longrightarrow Q(x) \\ R2: & P(A) \Longrightarrow Q(A) \\ R3: & P(A) \Longrightarrow Q(B) \\ R4: & P(A) \Longrightarrow Q(x)\,. \end{array}$$

The subgoal $P(x)$ subsumes the three subgoals $P(A)$ with the bindings $c = \{x : A\}$, but not all of these subgoals are redundant with $P(x)$. The leftmost $P(A)$ in Figure 3-8 is redundant because its binding set $\{x : A\}$ is identical to c. However, the second $P(A)$ has the binding set $\{x : B\}$, which does not contain $x : A$. Therefore, this subgoal is not redundant with the subgoal $P(x)$, unless x is not an output variable. The third instance of $P(A)$ is also not redundant with the subgoal $P(x)$ (assuming x is an output variable), since it has an empty binding set. Finally, it is worth noting that the first and second instances of $P(A)$ are redundant with the third instance of $P(A)$. For these cases c is empty and therefore, is contained in any binding set.

Descendant Subsumption We can apply Theorem 3.8 to cases of repeating inference. In this case, g'' is a descendant of g', and the root goal g can be taken to be g'. Thus, the binding list b' is empty.

Corollary 3.9 *Let g' be a repeating descendant of g and let b be the set of bindings that relate solutions to g' to solutions to g ($g' \Longrightarrow g|_b$). Let c be the set of bindings such that $g' = g|_c$. If the bindings for the output variables in c are contained in b, the goal g' is redundant with the goal g. Furthermore, the optimal strategy does not involve g'.*

As an example, consider the simple search space of Figure 3-9 generated from the goal $P(x)$ and the rules

$$\begin{aligned} R1: &\ P(A) \Longrightarrow P(A) \\ R2: &\ P(A) \Longrightarrow P(B) \\ R3: &\ P(A) \Longrightarrow P(x)\,. \end{aligned}$$

The three subgoals $P(A)$ are subsumed by the root goal $P(x)$ with the

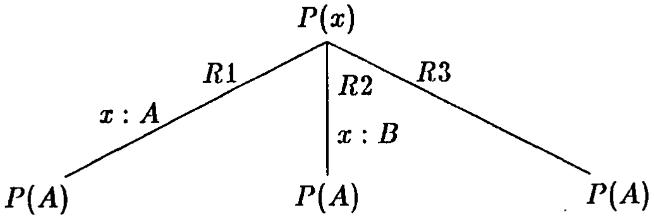

Figure 3-9: Repetition subsumption example

binding set $c = \{x : A\}$. The first subgoal has bindings $b = \{x : A\}$ and can therefore be eliminated. The second has the set of bindings $b = \{x : B\}$, which does not contain c. Therefore, it cannot be eliminated if x is an output variable. Likewise, the third subgoal cannot be eliminated since its binding set b is empty.

The most common cases of descendant subsumption are when a descendant is identical to an ancestor in every respect, including variables. For this case, the set c is empty and the descendant can be eliminated. These situations arise from if-and-only-if rules expressing definitions and from rules expressing properties like symmetry, associativity, and commutivity. For example, in a circuit analysis system we might need the information that

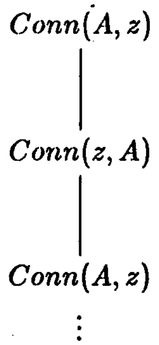

Figure 3-10: Search path for a symmetry rule.

electrical connections are symmetric.

$$Conn(x, y) \Longrightarrow Conn(y, x)$$

Suppose we were to apply this rule for the problem of finding all the points in a circuit connected to a given point A.

$$\textit{find all } z\text{: } Conn(A, z)$$

We first get the subgoal $Conn(z, A)$. Applying the rule to this subgoal gives the subgoal $Conn(A, z)$ again, as shown in Figure 3-10. This is silly, since it only leads us back to where we started. We can always prune such repetition according to the subsumption theorem.

Transitivity The subsumption theorem also has a direct application to repeating inference resulting from transitivity rules. Consider the connectivity example used in the previous sections, with the query $Conn(A, z)$ and a database containing the facts

$$\begin{array}{l} Conn(A, B) \\ Conn(B, C) \\ Conn(C, D)\,. \end{array}$$

Figure 3-11 shows the portion of the space that would be generated for this problem using Algorithm 3.7. First, the answer $z = B$ is found in the database. Then, the conjunctive subgoal $Conn(A, y) \wedge Conn(y, z)$ is generated. The first of these conjuncts is repeated, so we plug in the answer

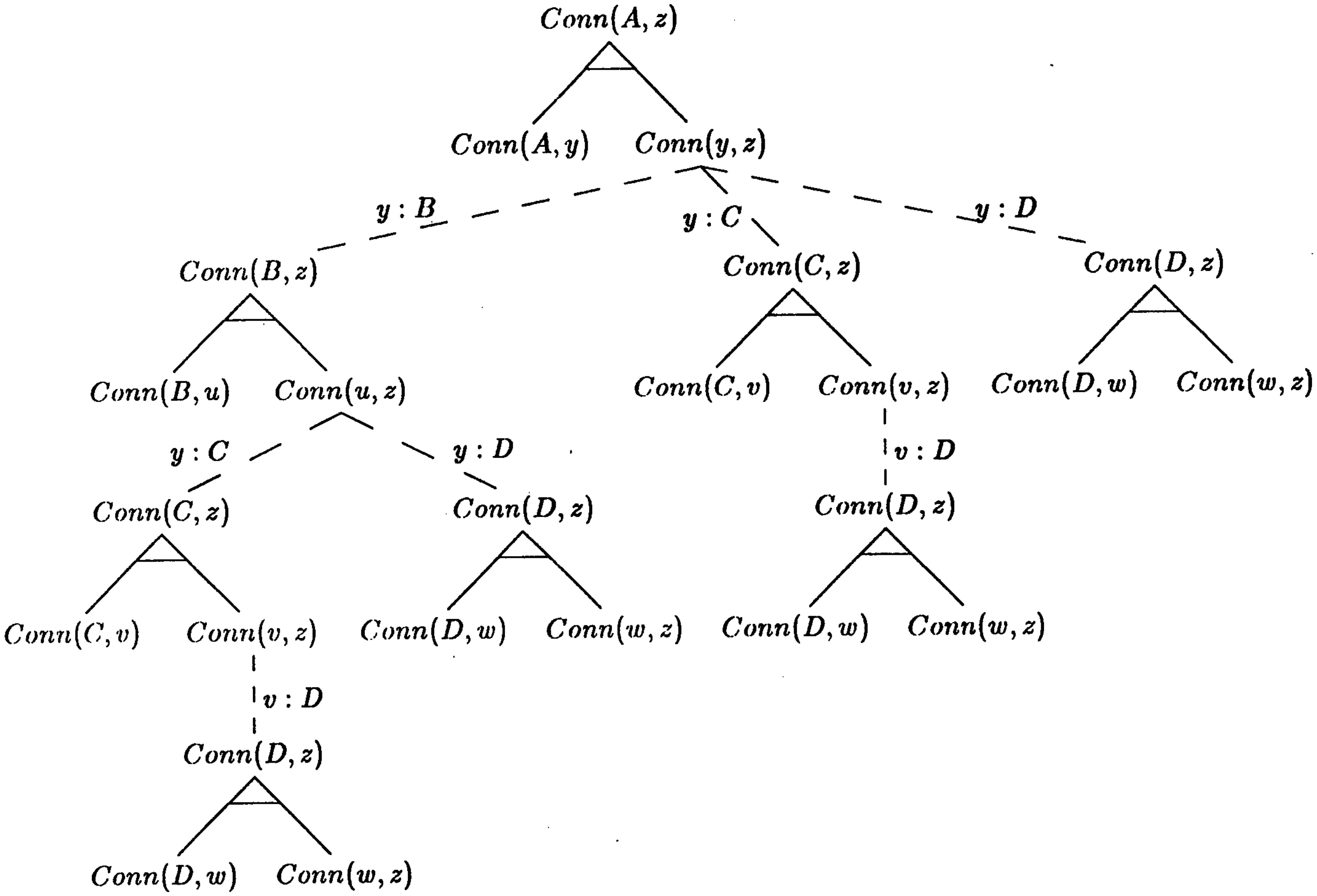

Figure 3-11: Search space for the query $Conn(A, z)$.

$x = B$ that has already been found. Continuing on the remaining conjunct $Conn(B,z)$ we find one answer in the database, $z = C$, and use the transitivity rule to generate the subgoal $Conn(B,u) \wedge Conn(u,z)$. The first of these conjuncts is a repeat of its parent so we again plug in the solution already found, $u = C$. The remaining conjunct becomes $Conn(C,z)$. Again, there is a single answer in the database, $z = D$. We apply the transitivity rule one more time yielding the conjunction $Conn(C,v) \wedge Conn(v,z)$. The first of these is again a repeat of its parent and we plug in the available solution, $v = D$. The remaining conjunct, $Conn(D,z)$, yields no solutions, so we begin to unwind. Note that we have already found all of the solutions to the problem, $z = B, C$ and D, yet we have not substituted the newly generated answers into the two remaining repeating descendants. According to the algorithm, we must substitute $u = D$ into the conjunction $Conn(B,u) \wedge Conn(u,z)$. Following this, we must substitute the answers $y = C$ and $y = D$ into the first subgoal $Conn(A,y) \wedge Conn(y,z)$. Each of these substitutions causes more redundant inference. In effect the procedure produces each of the answers twice. A similar situation occurs with the dual query $Conn(x,D)$ (assuming the conjuncts are processed in a reasonable order) and with the general query, $Conn(x,z)$.

Much of the duplication can be eliminated by recognizing and pruning subsumed goals. For example, the two instances of the goal $Conn(C,z)$ are mutually redundant according to the subsumption theorem. Likewise, the four instances of the goal $Conn(D,z)$ are mutually redundant. If all but one of each are eliminated the remaining search space does not contain any redundant portions. Using the subsumption theorem, together with Algorithm 3.7 therefore solves the problem. However, the two results can be combined into a more succinct reduction theorem.

Theorem 3.10 *Let $g' \wedge g''$ be the conjunctive subgoal produced by applying a transitivity rule*

$$R(x,y) \wedge R(y,z) \Longrightarrow R(x,z)$$

to the goal g, as illustrated in Figure 3-12. Let h' and h'' be the conjunctive subgoals produced by the application of the transitivity rule to the conjuncts g' and g'' respectively. Then, h' and h'' are mutually redundant. In other words, $S(g) = S_{h'}(g) = S_{h''}(g)$.

Proof *For all possible g that match $R(x,z)$, the conjunction $g' \wedge h''$ subsumes $h' \wedge g''$ and vice versa. For example, if $g =$ "$R(x,z)$", $g' =$ "$R(x,y)$",*

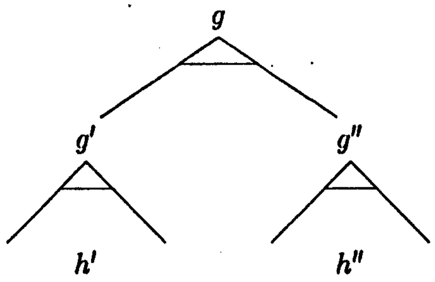

Figure 3-12: Transitivity search space.

g'' = "$R(y,z)$", h' = "$R(x,v) \wedge R(v,y)$" and h'' = "$R(y,w) \wedge R(w,z)$", then $R(x,y) \wedge R(y,w) \wedge R(w,z)$ subsumes $R(x,v) \wedge R(v,y) \wedge R(y,z)$ and vice versa for any subset of $\{x,z\}$ as output variables. The theorem therefore follows immediately from the subsumption theorem. □

This result is easily implemented. When a transitivity rule is applied to a goal, the rule should not be applied to one of the two conjunctive subgoals generated. For the connectivity example the two possibilities are shown in Figures 3-13 and 3-14. If the transitivity rule is not reapplied to left-hand branches the result is a simple, but lopsided search space. If it is not reapplied to the right hand branch, repeating inference occurs in the left-hand branch, and the methods of Section 3.2 must be applied. Using Algorithm 3.7, inference on the left-hand branch would stop after one level. All answers are generated merely by caching solutions and substituting them into the left branch.

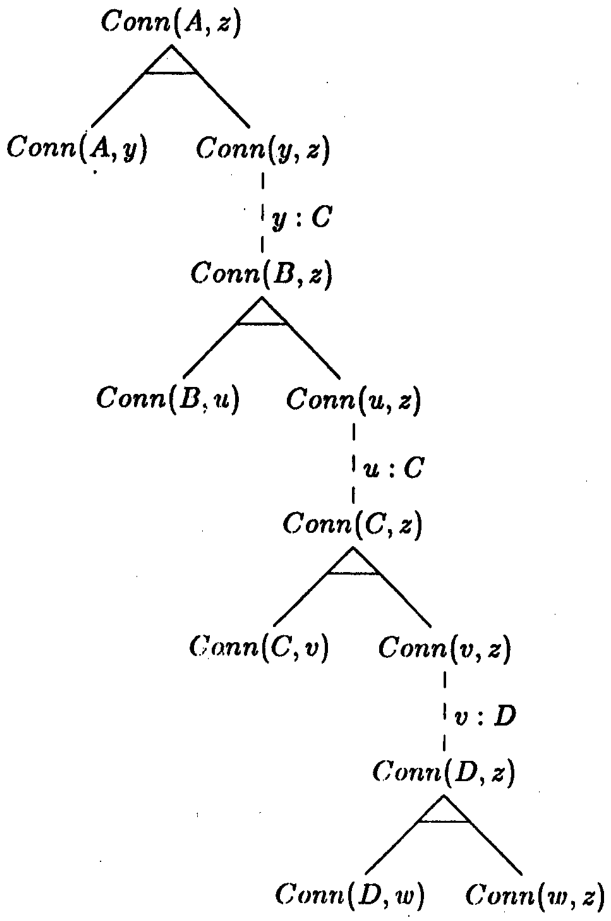

Figure 3-13: Left-pruned search space for the goal $Conn(A, z)$

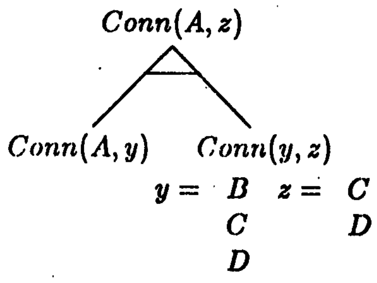

Figure 3-14: Right-pruned search space for the goal $Conn(A, z)$.

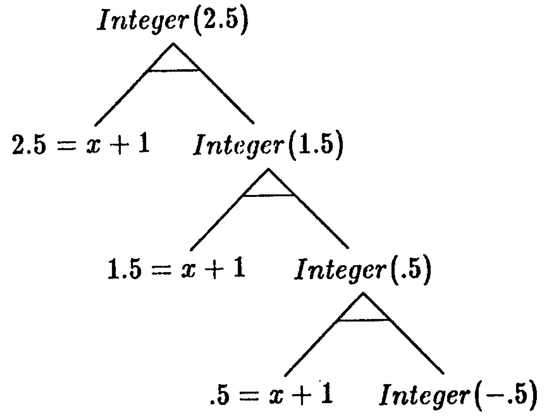

Figure 4-1: Search space for the query *Integer*(2.5). The first conjunct has been evaluated for each subgoal.

4 Divergent Inference

The most troublesome form of inference loops are those that do not repeat. Consider again the simple rule describing the integers:

$$y = x + 1 \ \wedge \mathit{Integer}(x) \Longrightarrow \mathit{Integer}(y) \qquad (4-1)$$

A query such as *Integer*(2.5) generates an infinite sequence of subgoals like that shown in Figure 4-1. If we were to list the sequence of goal conjuncts reduced at each step in the inference process, no specific conjunct would appear more than once in this sequence. There is an infinite number of *Integer* conjuncts in this sequence, but each one has a different argument. As we indicated in Section 1.3 we refer to such non-repeating recursive inference as *divergent* inference.

How do we go about cutting off inference in such a case? In general it is only semi-decidable whether or not the space below a given subgoal will contain novel answers to the problem. Yet for a case like the one above we can supply a fairly simple argument for pruning the infinite recursion from the search space. Suppose that the smallest integer in the database is 2. The sequence of descendants from *Integer*(2.5) is monotonically decreasing. As a result, once we have passed *Integer*(2) all further descendants are superfluous; they will never be able to match any fact in the database. This argument is not unlike the sort of arguments used in proving program correctness or program termination. .Here we have used, as an invariant assertion, the fact that every descendant of $Integer(2.5)$ will be of the form $\phi(x) \wedge Integer(x)$ and that x will always be less than 2.

This kind of argument can be generalized to arbitrary recursive collections. What is necessary is to find an invariant assertion for each goal form in the loop that implies that there will be no answers in the database for the corresponding goal. In addition we must show that all other rules that apply to goals in the loop (rules not in the recursive collection) will not produce any answers.

We can make this kind of argument more precise. Let the relation $No(p)$ mean that there are no facts in the database that unify with the proposition p.

Theorem 4.1 *Let $\{R_1, \ldots R_m\}$ be the relations occurring in the consequents of the rules in a cyclic collection (recursive collections included). Let*

$$F_{j,k,n} = \text{“}\phi_{j,k,n}(\vec{y}, \vec{z}) \wedge R_j(\vec{y}) \Longrightarrow R_k(\vec{z})\text{”}$$

designate the n^{th} rule in the collection having a relation R_j in its premise, and R_k in its consequent. (The $\phi_{j,k,n}$ may contain additional R from the set.) Suppose that there is a predicate β_k on the domain of each relation R_k such that

1. $\beta_k(\vec{y}) \Longrightarrow No(\text{“}R_k(\vec{y})\text{”})$,
2. $\beta_k(\vec{z}) \wedge \phi_{j,k,n}(\vec{y}, \vec{z}) \Longrightarrow \beta_j(\vec{y})$, *and*
3. $\beta_k(\vec{z}) \Longrightarrow Superfluous(\gamma)$, *for all other facts,* $(\gamma \Longrightarrow R_k(\vec{z}))$, *not in the recursive collection.*

Then, if $\beta_k(\vec{A})$ holds, the goal $R_k(\vec{A})$ is superfluous.

Here $\beta_k(\vec{z})$ is the invariant assertion for those goals with the relation R_k. The first condition states that the invariant assertion assures that no answers will be found. The second condition states that the invariant assertions are preserved from a goal to its immediate subgoals, and the third condition assures that none of the exit points of the loop will lead to any answers.

Proof *First we consider just those descendants of $R_k(\vec{A})$ that can be generated using rules in the cyclic collection. We want to show that there are no answers in the database for any of these descendants. We know that each of these descendants will contain a clause $R_j(\vec{y})$ for some R_j in the set of* *relations described in the theorem. If we can show that the invariant assertion $\beta_j(\vec{y})$ holds for that descendant, condition (1) in the theorem tells us that there will not be any answers in the database. Thus, we want to show that, for each descendant g' generated using only the cyclic collection, there is some expression $\psi(\vec{y},\vec{z})$ and some relation R_j in the set so that g' takes the form*

$$g' = \text{``}\psi(\vec{y},\vec{z}) \wedge R_j(\vec{y})\text{''} \tag{4-2}$$

and that

$$\psi(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y}) \, . \tag{4-3}$$

We prove this by induction on descendant depth. For the initial goal $R_k(\vec{A})$, the induction hypothesis (4-2) holds if we let ψ be the empty clause, $\vec{y} = \vec{A}$ and let $j = k$. Likewise, (4-3) follows from the given, $\beta_k(\vec{A})$.

Now assume (4-2) and (4-3) for every l^{th} level descendant of the goal $R_k(\vec{A})$. Any $(l+1)^{st}$ level descendant will be a subgoal of some l^{th} level descendant. There are two possible ways of obtaining subgoals from an l^{th} level descendant $\psi(\vec{y},\vec{z}) \wedge R_j(\vec{y})$.

1. *Apply some rule to a clause of ψ. In this case the new subgoal will be of the form $\psi'(\vec{y},\vec{z}) \wedge R_j(\vec{y})$, which satisfies (4-2). Furthermore, since $\psi'(\vec{y},\vec{z}) \Longrightarrow \psi(\vec{y},\vec{z})$ and $\psi(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$ we get $\psi'(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$, which proves the second half of our induction hypothesis (4-3).*

2. *Alternatively, we could apply some rule $F_{i,j,n}$ to the clause $R_j(\vec{y})$. In this case the new subgoal will be $\psi(\vec{y},\vec{z}) \wedge \phi_{i,j,n}(\vec{x},\vec{y}) \wedge R_i(\vec{x})$. If we let $\psi'(\vec{x},\vec{z}) = \phi_{i,j,n}(\vec{x},\vec{y}) \wedge \psi(\vec{y},\vec{z})$ our subgoal becomes $\psi'(\vec{x},\vec{z}) \wedge R_i(\vec{x})$, which again satisfies (4-2). Furthermore, since $\psi(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$ and $\beta_j(\vec{y}) \wedge \phi_{i,j,n}(\vec{x},\vec{y}) \Longrightarrow \beta_i(\vec{x})$ (condition (2) of the theorem) we get that $\phi_{i,j,n}(\vec{x},\vec{y}) \wedge \psi(\vec{y},\vec{z}) \Longrightarrow \beta_i(\vec{x})$, or $\psi'(\vec{x},\vec{z}) \Longrightarrow \beta_i(\vec{x})$. Thus the second part of the induction hypothesis also holds.*

The hypotheses (4-2) and (4-3) therefore hold for all $(l+1)^{st}$ level descendants of $R_k(\vec{A})$ and by induction, for all descendants of $R_k(\vec{A})$ produced using only rules in the cyclic collection. It follows from condition (1) in the theorem that there will not be any answers in the database for any of these descendants.

What remains is to consider those descendants produced using rules not in the cyclic collection. Every such descendant will involve either applying such a rule directly to the goal $R_k(\vec{A})$, or to one of the goals $\psi(\vec{y},\vec{z}) \wedge R_j(\vec{y})$ generated using the cyclic collection. Again there are two ways of producing subgoals to a goal of the form $\psi(\vec{y},\vec{z}) \wedge R_j(\vec{y})$.

1. *Apply some rule to a clause of ψ. As before, such a subgoal will still satisfy the induction hypothesis and the previous argument holds.*
2. *Apply a rule, $\gamma \Longrightarrow R_j(\vec{y})$ to the clause $R_j(\vec{y})$ to yield the subgoal $\psi(\vec{y},\vec{z}) \wedge \gamma$. But by the induction hypothesis we know that $\psi(\vec{y},\vec{z}) \Longrightarrow \beta_j(\vec{y})$. By condition (3) of the theorem we conclude that γ is superfluous. Thus there are no answers in the database to any of the descendants of this subgoal.*

Therefore, there are no answers in the database for any of the descendants of $R_k(\vec{z})$, which means it is superfluous. □

4.1 Example

To see how this theorem applies, consider the simple integer example introduced earlier. For this example there is only one rule in the recursive collection. The relation in its consequent (R_1) is the *Integer* relation, and its ϕ will be $\phi_{1,1,1}(x,y) =$ "$y = x + 1$". If we choose $\beta_1(x) =$ "$x < 2$", the condition $\beta_1(y) \wedge \phi_{1,1,1}(x,y) \Longrightarrow \beta_1(x)$ will be satisfied. If the smallest integer in the database is 2, $\beta_1(x) \Longrightarrow No(R_1(x))$ is true. The final condition, that all rules not in the recursive collection result in superfluous goals, is true since there are no other rules. Then, according to the theorem $\beta_1(x) \Longrightarrow Superfluous(\text{"}R_1(x)\text{"})$, or $x < 2 \Longrightarrow Superfluous(\text{"}Integer(x)\text{"})$. Therefore, we conclude that the subgoal *Integer*(1.5) is superfluous.

4.2 Application of the Theorem

In general, mechanizing the application of Theorem 4.1 is not a simple matter. First we choose an applicable recursive collection to apply the theorem to. It may be that all recursive collections are already known and marked in the database. In this case finding an applicable recursive collection is a straightforward lookup operation. If not, we must recursively enumerate the set of applicable rules looking for recursive collections. This is done by mapping through each rule that applies to a goal and doing the same for each subgoal. If the same rule is used again in any path, a cycle and possible recursive collection has been located. If several independent recursive collections are found we must choose one. In satisfying the final criterion of the theorem, that all other applicable rules do not result in any answers, the others will be considered (the theorem may need to be applied recursively to prove these cases).

Ambiguity in choosing the recursive collection also arises when two or more recursive collections share a common rule. In this case we have nested or interwoven loops. According to the definition of a recursive collection, their union also constitutes a recursive collection. We could therefore choose to apply the theorem to one of the individual recursive collections, or to the composite collection. If we choose to apply it only to an individual collection, in the final step it will be necessary to prove that none of the other interwoven loops can yield an answer. This is usually more difficult. It is therefore probably wise to consider the maximal recursive collection first.

The second step is to collect the set of consequent relations in the recursive collection. This is straightforward.

The third step is to find a set of invariants β_i that satisfy the characteristics

1. $\beta_k(\vec{z}) \Longrightarrow No(``R_k(\vec{z})")$, and
2. $\beta_k(\vec{z}) \wedge \phi_{j,k,n}(\vec{y}, \vec{z}) \Longrightarrow \beta_j(\vec{y})$.

This task involves generating possible predicates β_i and testing them to see if they satisfy the above axioms. The most efficient way to do this is to start at one place in the loop, and proceed around the loop in an orderly fashion, generating the β at each step. Thus we start by generating a possibility β_k for some k and check to see that it satisfies the first axiom above. Then, choose j so that there is some rule $F_{j,k,n}$. Now generate β_j and check to see that it satisfies both the first and second axioms. Then, choose i so that there is some rule $F_{i,j,n}$ and so forth.[6]

The real problem is in generating good candidates for any individual β_i, particularly since the desired β_i might be a conjunction of known relations. We could start by considering all known predicates on the domain D_i of R_i. If none of these work, we could try all known relations from D_i to a new domain D' and conjoin these with known predicates on D'. If none of these work, we consider conjunctions containing three relations, and so on. In general this may be necessary. However, it seems that β often takes the form of an integer bound,

$$\beta_i(\vec{x}) = ``\gamma(\vec{x}, l) \wedge l \sim N"$$

[6] Manna and Waldinger [MW77] discuss more sophisticated ways of generating loop invariants for the purposes of program verification. Much of this work appears to be applicable here.

where $\sim$ is one of $<, >$ *or* $=$ and N is a fixed integer. In our simple integer case, γ was the identity relation. However, if we were dealing with lists of ever increasing length, the *Length* function might be appropriate. Similarly, if we were dealing with human ancestry a function such as *Birthdate* might be appropriate. The strategy for generating β_i therefore involves first considering an empty γ if the domain of R_i is the integers. If this fails, known relations mapping the domain D_i onto the integers are considered. In effect, this is a way of generating possible ordering relations for domains where an ordering relation is not already known. Although it may, in theory, be necessary to consider conjunctions of known relations for γ, if the number of known relations is large, the space of possibilities quickly becomes intractable when more complex γ are considered.

The final step involves verifying that none of the other relevant facts will generate any answers to the problem. This may be trivial as in the integer example, or it can be arbitrarily difficult, if there are other recursive collections involved. In the latter case, this final step may well involve application of Theorem 4.1 or any one of the cutoff theorems developed in Section 3.

4.3 Functional Embedding: A Special Case

A common cause of divergent inference are rules that contain functional expressions on their left hand sides. By this we mean rules of the form,

$$P(F(x)) \wedge \phi \Longrightarrow P(x) .$$

For example, the rules,

$$\begin{array}{c} Jewish(Mother(x)) \Longrightarrow Jewish(x) \\ Integer(Successor(x)) \Longrightarrow Integer(x) \end{array}$$

are both of this sort. Such rules will always lead to divergent inference if the inference engine does not evaluate the embedded functional expressions. For example, a query such as $Jewish(Job)$ would lead to the subgoals $Jewish(Mother(Job))$, $Jewish(Mother(Mother(Job)))$, etc.

For such cases we can often choose β to be a lower bound on the level of functional embedding in a subgoal. If there are no rules relevant to a problem that can shrink the amount of functional embedding (e.g. $P(f(f(x))) \Longrightarrow Q(X))$, it is possible to stop the inference process whenever the level of functional embedding exceeds the largest embedding available in the database. For example, in the jewish ancestry problem, if the fact with the largest functional embedding is $Jewish(Mother(Mother(Job)))$, any subgoal having a functional embedding deeper than two could be discarded. Notice that this strategy refers to total functional embedding independent of the actual functions involved. The reason is that there may be rules available such as $P(G(x)) \Longrightarrow P(F(x))$ that can result in new subgoals having different embedded functions.

4.4 Commutivity of Inference Steps

In the previous section we considered only cases where it was possible to prove that no answers existed in a portion of the search space. In fact we can generalize Theorem 4.1 by only insisting that subgoals contribute no novel answers to the overall problem (as opposed to no answers at all). It is usually quite difficult to prove that answers generated somewhere in a loop will not result in novel answers to the overall goal. However, there are special cases where redundancy can be proven, and in such cases Theorem 4.1 can be applied. In this section we develop such a special case result for situations where the inference steps are *commutative.*

Consider the pair of axioms:

$$\begin{aligned} R_1: \ & y = x + 1 \wedge Integer(x) \Longrightarrow Integer(y) \\ R_2: \ & y = x - 2 \wedge Integer(x) \Longrightarrow Integer(y) \end{aligned}$$

As before, suppose that the problem is to determine whether or not 2.5 is an integer, and the smallest integer in the database is 2.

When only the first of the above two rules was available, we argued that the sequence of subgoals from $Integer(1.5)$ was monotonically decreasing, and therefore the subgoal $Integer(1.5)$ was superfluous. Given both rules, this argument no longer holds, since $Integer(3.5), Integer(5.5), \ldots$ are now descendants of the subgoal $Integer(1.5)$. In fact, imagine that the fact $Integer(5.5)$ happened to be in the data base. Then, the subgoal $Integer(1.5)$ would not be superfluous, since the problem could be solved by exploring one of its descendants.

Even though the goal $Integer(1.5)$ may not be superfluous, we can argue that it is redundant with its supergoal, $Integer(2.5)$. The argument depends on the observation that any application of the two rules above is *commutative.* In other words, if a subgoal can be produced by applying one rule, then the other, it can also be produced by applying the rules in the reverse order. For example, the subgoal $Integer(3.5)$ can be produced from the goal *Integer*(2.5) by applying R_1 followed by R_2. Alternatively, it can be produced by applying R_2 followed by R_1.

Using this observation, we separate the descendants of *Integer*(1.5) into two groups, those generated by applying only the first of the two rules, and those that involve at least one application of the second rule. From our earlier argument we know that the first class will not result in any answers. For the second class, since the two rules are commutative, the same subgoal can be produced by first applying the second rule to the goal *Integer*(2.5). Thus the subgoal *Intcgcr*(1.5) is redundant.

We now make this kind of argument precise. We say that two sets of rules are commutative if any subgoal produced using a rule from one set followed by a rule from the other set could also be produced by using the rules in the opposite order. More formally,

$$\begin{aligned}&Commutative(s,t) \Longleftrightarrow \forall q \in s, r \in t, g, g', h\\&\quad (Subgoal_q(g,g') \wedge Subgoal_r(g',h)\\&\qquad \Longrightarrow \exists g'' Subgoal_r(g,g'') \wedge Subgoal_q(g'',h))\end{aligned}$$

where the notation $Subgoal_r(g,h)$ means that the goal h can be derived as a subgoal of the goal g using the rule r.

Theorem 4.2 *Suppose that the set of applicable facts for a goal g can be broken up into two commutative subsets s and t. Suppose that all descendants of g, generated using only rules in s, produce no novel answers (i.e. if only rules in s are used, the subgoal g would be redundant). Then, all (immediate) subgoals of g produced using rules in s are redundant for the entire collection $s \cup t$.*

Proof *Let g' be an (immediate) subgoal of g generated using a rule from the set s. Consider an arbitrary descendant, d, of g'. Suppose d is produced using only the rules in the set s. Then, by the premises of the theorem there are no answers in the database for d that result in novel answers to the overall problem. Alternatively, suppose that d is produced using, at least one rule r from the set t. Since the rules in s and t are commutative, the descendant d will also be a descendant of the subgoal g'' generated by applying r to the goal g. All descendants of g' (including g' itself) are therefore redundant with immediate subgoals of g produced using rules in t. Since our choice of g' was arbitrary all immediate subgoals of g produced using rules in s are redundant.* □

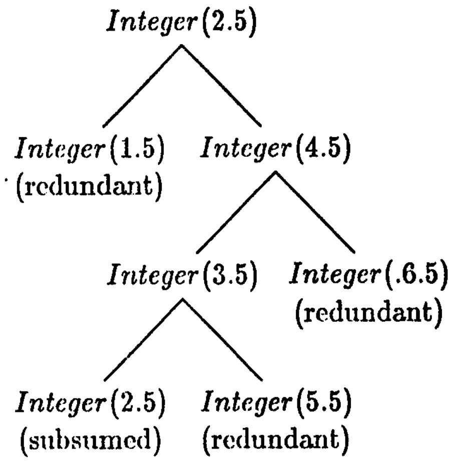

Figure 4-2: Search space for the query *Integer*(2.5). The first conjunct has been evaluated and removed from each subgoal

4.5 Example

Using the theorem above, together with the results of previous sections, the two rule integer example can now be handled.

There are two immediate subgoals of the goal *Integer*(2.5). The first, *Integer*(1.5) is redundant according to Theorems 4.2 and 4.1. Therefore, it is possible to eliminate it without sacrificing any answers. The other subgoal, *Integer*(4.5), has the two immediate subgoals *Integer*(3.5) and *Integer*(6.5). If the maximum integer in the database is 3, we can again apply Theorems 4.2 and 4.1 to show that *Integer*(6.5) is redundant. The other subgoal, *Integer*(3.5) has the two immediate subgoals *Integer*(2.5) and *Integer*(5.5). The latter is again redundant, by application of Theorems 4.2 and 4.1. The remaining subgoal *Integer*(2.5) is identical to the original goal, and so by Theorem 3.9 it can also be eliminated. The abbreviated subgoal tree is shown in Figure 4-2.

It is interesting to note that if we changed the constant in either of the two rules to an irrational number, the inference could not be completely stopped. We could still apply Theorems 4.2 and 4.1 at each level of the search space, but we would never return to the original goal, and therefore could never apply Theorem 3.9. In this case, the chain of subgoals would continue to bounce back and forth between the two extreme integers in the database.

4.6 Remarks

In this section, we first developed a general theorem for terminating divergent inference, and used it to solve a simple problem involving a single loop. We also applied the theorem to the special case of functional embedding.

It is generally more difficult to apply the theorem to cases of redundancy, as we can see from the two rule integer example above. Powerful special case methods like Theorem 4.2 seem essential for dealing with such complex cases. We have investigated only one such result here. Additional work is needed to build up a library of theorems, like 4.2, that can be used for various difficult cases of divergent inference.

5 Discussion

5.1 Detecting Recursive Inference

There is always a price to pay for controlling inference and control of recursive inference is no exception. For repeating inference the cost is relatively low. It involves suspension of repeating goals and the caching of answers to goals with repeating subgoals. However, for divergent inference, control involves explicit proofs that subspaces are superfluous or redundant. When the alternative is an infinite loop, any finite control cost is justifiable. The problem is, we usually cannot be certain whether or not an infinite loop will result. As we pointed out in Section 2, even when a problem has an infinite recursive search space, recursive inference will not necessarily occur. The necessary answers might be found before an infinite path is explored by the inference engine.

In general, it is undecidable whether or not a given inference procedure will terminate when searching a recursive space.[7] The best that can be done is to institute control only when it is considered likely that it will be necessary or cost-effective. This general issue is discussed further in [Smi85]. For recursive inference there are several interesting strategies. The simplest is to monitor search depth or total search space size and institute control when it exceeds some threshold. A more elaborate, but more costly scheme is to preserve the subgoal and justification trees and institute control when a given fact has been used more than some fixed number of times in the derivation of a particular subgoal. A third alternative is to limit control to those cases where a recursive collection is involved in the deduction. (Recursive collections could be recognized either when rules are entered into the system, or during the problem solving process.)

Each of the strategies has certain advantages and disadvantages. In general, they trade accuracy for expense. For example, the recognition of rule reuse is usually a more accurate predictor of recursive inference than overall search depth, but it is also more expensive since it requires keeping the goal and justification stacks and searching them for each new subgoal.[8] As a result, the best strategy for a given application will depend upon such

[7] The problem is equivalent to the halting problem for Turing machines since backward inference over a set of axioms is Turing equivalent.

[8] Associating a marker or counter with each rule doesn't work in general. The marker would have to be path dependent since we do not wish to count the repeated usage of rules in independent inference paths.

things as the average depth of inference, the frequency of recursive inference, and the density of recursive collections in the system's database.

There is also no reason why these strategies cannot be combined. For example, we could use search space depth or complexity to determine whether or not to initiate the strategy of checking for recursive collections or repeated rules. Likewise, the strategy of checking for repeated rules could be used as a filter for the strategy of looking for recursive collections. These combined strategies allow a less expensive but less accurate detection criteria to serve as a filter for a more accurate and more costly one. Such combinations may, in fact, prove to be the most cost-effective for many applications.

5.2 History and Related Work

5.2.1 Recursive Inference

Black [Bla68] and McKay and Shapiro [MS81] describe algorithms for stopping repeating inference similar to those developed in Section 3.2.2. However, they do not provide any proof that the pruning strategy is correct and do not consider the question of optimality. They also do not consider any of the special cases (like transitivity, subsumed subgoals, or single answer queries) where more powerful strategies can be used.

The special case of eliminating identical subgoals appears to have been first used by Gelernter in his geometry theorem proving program. [Gel63].

> *Subgoals ... are rejected ... that appear as higher subgoals on the [subgoal] graph (or are syntactically symmetric to some higher subgoal). page 142*

In Gelernter's application since all goals and subgoals are geometry theorems requiring only a yes or no answer both Theorems 3.1 and 3.9 apply. As a result, all repeated subgoals can be eliminated for this particular application.

Special cases of repeating inference were also encountered in building the MYCIN system [Sho84]. One cause of repeating subgoals in MYCIN was the use of *self-referencing* rules. A self-referencing rule states that if there is already evidence for a condition ψ and some other condition ϕ holds, there is additional evidence for the condition ψ. These rules therefore include the proposition ψ in both the premise and conclusion. MYCIN handles a self-referencing rule by postponing it until all other rules for concluding ψ have been used. Then, the self-referencing rule is applied exactly once.

This strategy involves pruning all repeated applications of a rule, a much stronger pruning strategy than is indicated by Theorem 3.3. This strategy works for self-referencing rules because they are actually quite different from recursive rules. Consider how we would translate a self-referencing rule into a precise declarative statement. We might be inclined to write something like

$$\psi \wedge \phi \Longrightarrow_p \psi$$

where $\Longrightarrow_p$ means that we have p additional evidence for the conclusion (the actual calculus for combining certainties is unimportant). If this statement were true, given some small amount of evidence for ψ, and the fact that ϕ is true, we could use this axiom over and over again to derive greater and greater belief in ψ.[9] This is not the intended meaning of the self-referencing rule. In a self-referencing rule the recursive premise is a screen to prevent exploration of ϕ unless there is already some evidence of ψ. In other words, the recursive premise is control knowledge about when to apply the simpler rule

$$\phi \Longrightarrow_p \psi \, .$$

Thus a logical translation of a self-referencing rule would consist of two rules; the simple rule given above, and a control rule indicating that the above rule should not be tried unless there is already evidence for ψ. Neither of these rules are recursive so the theorems developed in Section 3 do not apply. This translation also shows why MYCIN's pruning strategy for self-referencing rules is appropriate. Since the above rule is not recursive, it need be applied only once.

Repeating inference also occurs in MYCIN as a result of rule loops. For example, a rule might allow a parameter B to be inferred from a parameter A, while another rule might allow A to be inferred from B. MYCIN's strategy in such cases is to never use a rule more than once in a single reasoning chain. This is equivalent to pruning all repeating subgoals. This works because, once the context is bound, the premises and conclusions of such rules are ground clauses. Thus, as in Gelernter's application, the powerful pruning strategy of Theorem 3.9 applies.

More recently, Minker and Nicolas [MN83] have developed a special case of Corollary 3.9 and have shown that for the class of "singular" recursive rules all repeating subgoals will be subsumed and can therefore be eliminated.

[9] It is interesting to note that, Theorem 3.1, as stated, will not hold in the case of uncertain reasoning. Even if ψ is a ground clause, recursion could continue to increase the belief in ψ. However, from an evidential point of view we would never want to allow this, since arguments should not be circular.

Other approaches to controlling repeating inference have also received some attention, although the results have been limited to special cases. Reiter [Rei78] and Minker and Nicolas [MN83] have shown conditions where it is possible to use only forward inference on recursive collections. Automatic reformulation of recursive collections has been explored by Chang [Cha80] and Naqvi and Henschen [NH80]. They describe methods of automatically generating efficient procedures for the special class of "regular" recursive collections. Minker and Nicolas [MN83] have shown that this method applies to a slightly broader class of recursive collections. A somewhat different approach to control is that advocated by Ullman [Ull84]. Ullman first analyzes the search space for queries of a given form (given relation and given set of bound and free variables) to find potential recursive paths. He then uses this information to avoid such paths during the reasoning process.[10] The most serious problem with this approach is that it can result in incompleteness for queries that require some amount of recursive inference. It is also not clear that the approach holds any advantage over algorithms like those in Section 3.2.2.

The more difficult problem of divergent inference has received little attention in the literature. Fischer Black noted the problem in developing his natural deduction system [Bla68] but provides no solution other than depth-limited search.

Problems of recursive inference have also arisen outside of the Artificial Intelligence community. Both repeating and divergent inference are constant obstacles in the construction of PROLOG programs. Users of PROLOG become well versed in manual reformulation of rules to eliminate infinite loops. As we illustrated, this is not always an easy task and the resulting programs can be quite opaque.

Recently, at Stanford we have encountered repeating and divergent inference in the construction of systems for reasoning about digital circuits [Gen85,Kra84]. The techniques described here are being implemented in an experimental version of the MRS system [Rus85].

5.2.2 Program Verification

There is a strong similarity between recursive inference and recursive programs that do not terminate. In essence, an inference procedure together with a goal and recursive collection of facts is a recursive program. Thus,

[10] The approach bears some resemblance to that for controlling backward inference in [Smi85]. In fact Ullman's approach also applies to more than just recursive inference.

it should come as no surprise that the techniques for deciding whether a given inference path terminates, loops, or diverges, bear a striking resemblance to the method of well-founded sets used to prove program termination [Flo67,Man74].

But the similarities end here. In general, a recursive program that does not terminate is of little use. It must be modified so that it does terminate. Doing this requires some understanding of the intention behind the program. In contrast, a recursive collection of facts has *meaning*, independent of the particular inference engine being used. In this case, the inference engine must be modified so that it will perform the proper deductions. The intent of the inference procedure is known. Thus, the change takes the form as information about how to prune the search space. In-short, a program that does not terminate may be incorrect in an arbitrary manner, while an inference engine that loops (for a given goal and recursive collection) is incorrect in a very specific way; it explores too much of the search space.

5.3 Final Remarks

The control of recursive inference involves demonstrating that portions of a search space are either superfluous or redundant. When either of these properties has been demonstrated, the offending portion of the search space can be discarded. Although this will always be logically correct, it may not be optimal in every case.

Proofs of redundancy and superfluity involve knowledge about the contents of the system's database, and about properties of the relations involved in the inference, such as ordering relations on the domains, monotonicity, boundedness and commutivity. This kind of information is commonly available but has rarely been needed or used in AI systems. In contrast to the general domain-dependent character of the control problem, the special case of repeating inference admits control, which is domain-independent. The method of suspending and reenabling repeated subgoals does not depend upon the meaning of the symbols involved. It is not entirely clear why this fortuitous result should hold. It is true however, that in some cases the more general domain-dependent techniques of proof can lead to more severe pruning for repeating inference than is possible with the syntactic method of Section 3.

Determining whether or not recursive inference will occur for a given problem is in general undecidable. We have suggested three possible criteria for determining when control of recursive inference should be instituted;

inference depth or complexity, repeated rule usage, and the use of recursive collections. Combinations of these approaches also appear promising, although the decision will almost certainly prove dependent upon the mix of problems encountered in any particular application.

Finally, there is no a priori reason why the techniques of proving redundancy and superfluity could not be applied to non-recursive inference. The limiting factor is cost. When an infinite search is avoided, a high cost is justifiable. However, for non-recursive inference the problem would have to be a difficult one for expensive analysis like that of Section 4 to be cost-effective. For such cases, complex monitoring strategies like those proposed in Section 5.1 would be indispensable.

Acknowledgements

Special thanks to Narinder Singh and Glenn Kramer for bringing interesting examples of repeating and divergent inference to our attention. Thanks to Russ Greiner, Jock Mackinlay, Vineet Singh, Richard Treitel and the other members of the Logic Group at Stanford for general discussion on this subject over the last several years. Jim Bennett, Jan Clayton and Ted Shortliffe provided information on self-referencing rules and loops in MYCIN and EMYCIN. Jeff Finger, Pat Hayes, and Richard Waldinger provided useful pointers to related work on program verification while Richard Treitel and Jeff Ullman provided pointers to related work on databases and logic programming. Thanks to Jan Clayton and Jock Mackinlay for proofreading assistance and comments on presentation.

This work was supported by ONR contract N00014-81-K-0004.

References

[Bla68] F. Black. A deductive question-answering system. In M. Minsky, editor, *Semantic Information Processing*, pages 354-402, MIT, Cambridge, 1968.

[Cha80] C. L. Chang. On evaluation of queries containing derived relations in a relational database. In *Advances in Data Base Theory*, pages 235-260, Plenum Press, New York, 1980.

[Cla83] W. J. Clancey. The epistemology of a rule-based expert system. *Artificial Intelligence*, 20(3):215-251, 1983.

[Dav80] Randall Davis. Meta-rules: reasoning about control. *Artificial Intelligence*, 15:179-222, 1980.

[Doy80] J. Doyle. *A Model for Deliberation, Action, and Introspection.* Artificial Intelligence Laboratory Memo AI-TR-581, Massachusetts Institute of Technology, May 1980.

[Flo67] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics*, pages 19-32, American Mathematical Society, Providence, R.I., 1967.

[Gel63] H. Gelernter. Realization of a geometry-theorem proving machine. In *Computers and Thought*, pages 134–152, McGraw-Hill, New York, 1963.

[Gen85] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1985.

[GS85] M. R. Genesereth and D. E. Smith. *Procedural Hints in the Control of Reasoning*. Knowledge Systems Laboratory Report HPP-84-11, Stanford University, January 1985.

[Hay73] P. J. Hayes. Computation and deduction. In *Proceedings of the Symposium on Mathematical Foundations of Computer Science*, pages 105–117, Czechoslovakian Acadamy of Sciences, 1973.

[Kra84] Glenn Kramer. December 1984. personal communication.

[Lew75] H. R. Lewis. *Cycles of Unifiability and Decidability by Resolution*. Technical Report, Aiken Computation Laboratory, Harvard University, 1975.

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw Hill, New York, 1974.

[McC68] J. McCarthy. Programs with commmon sense. In M. Minsky, editor, *Semantic Information Processing*, pages 403–418, MIT, Cambridge, 1968. Also as Proceedings of the Teddington Conference on the Mechanization of Thought Processes, Her Majesty's Stationery Office, London, 1960.

[MN83] J. Minker and J. Nicolas. On recursive axioms in deductive databases. *Information Systems*, 8(1):1–13, 1983.

[MS81] D. P. McKay and S. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh IJCAI*, pages 368–374, International Joint Conference on Artificial Intelligence, August 1981.

[MW77] Z. Manna and R. Waldinger. *Studies in Automatic Programming Logic*. North-Holland, New York, 1977.

[NH80] S. A. Naqvi and L. J. Henschen. Performing inferences over recursive data bases. In *Proceedings of the First National Conference* *on Artificial Intelligence*, pages 263–265, American Association for Artificial Intelligence, August 1980.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, 1980.

[Rei78] R. Reiter. On structuring a first order data base. In Perrault R., editor, *Proceedings Second National Conference*, pages 19–21, Canadian Soc. Computational Studies of Intelligence, Toronto, July 1978.

[Rus85] Stuart Russell. *The Compleat Guide to* MRS. Knowledge Systems Laboratory Report KSL-85-12, Stanford University, June 1985.

[Sho84] E. H. Shortliffe. Details of the consultation system. In *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, pages 78–132, Addison-Wesley, Reading, Mass., 1984.

[Smi85] D. E. Smith. *Controlling Inference.* PhD thesis, Stanford University, July 1985.

[Ull84] Jeffrey D. Ullman. *Implementation of Logical Query Languages for Databases.* Technical Report STAN-CS-84-1000, Stanford University, May 1984.